Tractable Reasoning in Knowledge Representation Systems[1]

Mukesh Dalal[2]

CUCS-017-95

Department of Computer Science
Columbia University
New York, NY 10027

July 1995

[1]This is an updated version of the Rutgers University Computer Science Technical Report DCS-TR-321 published in May 1995. This material is partially based upon work supported by the National Science Foundation under Grant No. IRI-94-10117.

[2]Email: dalal@cs.columbia.edu

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Examples 3

1.3 Plan 6

2 Fact Propagation **8**

2.1 Overview 8

2.2 Mathematical Preliminaries 9

2.3 PCE : Propositional Calculus with Equality and Generalized Connectives 9

2.3.1 Syntax 9

2.3.2 Semantics 12

2.3.3 Alternative Syntax for Examples 12

2.3.4 Directly Inferable Facts 13

2.3.5 Discussion 15

2.4 Rewrite Systems for PCE 15

2.4.1 Rewrite Systems 15

2.4.2 Properties of Rewrite Systems 18

2.4.3 Proof Techniques 19

2.5 BCP : Boolean Constraint Propagation 26

2.5.1 Horn Pebbling 26

2.5.2 Clausal BCP 28

2.5.3 Formula BCP and Prime BCP 29

2.5.4 CNF-BCP 30

2.5.5 Discussion 31

2.6 FPC : Extending Clausal BCP 31

2.6.1 Properties of FPC 33

2.7 FP : Extending CNF-BCP 35

2.7.1 Properties of FP 40

2.7.2 Comparison with CNF-BCP 41

2.8 FPE : Handling Equality 46

2.8.1 Properties of FPE 46

2.9 Conclusions . . . 49

3 Algorithms for Fact Propagation **50**

3.1 Overview . . . 50

3.2 Alternative Propagation Rules . . . 52

3.3 Basic Data Structures . . . 54

3.3.1 Tuple Encoding of Theories: . . . 59

3.4 AFPC: Simplification and Propagation Rules Only . . . 59

3.4.1 Correctness of AFPC . . . 66

3.4.2 Complexity of AFPC . . . 69

3.5 AFPL: Adding Lifting Rules . . . 72

3.5.1 Correctness of AFPL . . . 75

3.5.2 Complexity of AFPL . . . 76

3.6 AFP: Adding Factoring Rules . . . 78

3.6.1 Correctness of AFP . . . 84

3.6.2 Complexity of AFP . . . 85

3.7 AFPE: Adding Equality Rules to AFP . . . 88

3.8 Conclusions . . . 94

4 Deductions based on Fact Propagation **95**

4.1 Overview . . . 95

4.2 Preliminaries . . . 96

4.2.1 A Consequence Relation . . . 97

4.2.2 Basic Clauses . . . 98

4.2.3 Herbrand Bases . . . 99

4.2.4 Lattices and Fixed-Points . . . 100

4.3 Admissible Rewrite Systems . . . 100

4.4 A Fixed-point Construction for Viv . . . 106

4.4.1 The Fixed-Point Construction . . . 106

4.4.2 Properties of the Fixed-Point . . . 108

4.4.3 Vivid Theories and Intricacy . . . 109

4.4.4 Computing the Least Fixed-Point . . . 110

4.5 Consequence Relations for Limited Deduction . . . 111

4.5.1 A Complete Consequence Relation . . . 111

4.5.2 A Family of Tractable Consequence Relations . . . 112

4.5.3 Comparison with Earlier Approaches . . . 113

4.6 Conclusions . . . 115

5 Tractable Cases of Reasoning **116**

5.1 Overview . . . 116

5.2 Tractable Satisfiability Classes . . . 117

5.3 Tools for Proving Bounded Intricacy 120
5.4 Constraint Satisfaction Problems 123
5.4.1 Overview 123
5.4.2 Tractable Constraint Networks 124
5.4.3 Constraint Theories 129
5.4.4 Induced Width and Intricacy 132
5.4.5 Functional Constraints and Intricacy 133
5.4.6 A New Family of Tractable Classes 135
5.5 Databases with Disjunctive Information 140
5.5.1 Terminology and Notation 140
5.5.2 Query Answering by Checking Satisfiability 141
5.5.3 Tractable Algorithms for Querying 143
5.5.4 Describing Tractable Queries Using Intricacy 144
5.6 Disjunctive Logic Programming 148
5.6.1 A Tractable Family of DLPs 148
5.7 Conclusions 150

6 Conclusions **152**
6.1 Contributions 152
6.2 Directions for Future Research 153
6.3 Summary 154

A Axiomatic Proof Theory for PCE **155**

B A Ptime CNF Transformation **156**

C Local Consistency in CSP **158**

D Case Analysis for Proving Confluence **160**

List of Tables

3.1 Values of fields of nodes in the tree for $[\![P, ((\neg P \wedge Q) \vee P)]\!]$ 56

D.1 Summary of confluence case 161
D.2 Simplification and propagation rules 162
D.3 Simplification, propagation, and lifting rules 163
D.4 Lifting and simplification rules 164
D.5 Lifting and propagation rules 165
D.6 Propagation and lifting rules 166
D.7 Factoring and equality rules : $m, m', m'' \geq 1$ 167
D.8 Factoring and simplification rules: $m, m', m'' \geq 1$ 168
D.9 Factoring and propagation rules I: $m \geq 1$ 169
D.10 Factoring and propagation rules II: $m \geq 1$ 170
D.11 Propagation and factoring rules 171
D.12 Factoring and lifting rules 172
D.13 Equality rules I 173
D.14 Equality rules II: $m > 0$ and $a \succ b \succ c$ 174

List of Figures

1.1 Comparing FP and CNF-BCP . . . 3
1.2 FP on a constraint network . . . 5
1.3 Vivid transformation and Intricacy . . . 5

2.1 Logical term as a tree . . . 16
2.2 Overlap, critical pair, common term, and confluence . . . 23
2.3 Variable overlap: v is a meta-variable in R_1's schema . . . 24
2.4 A pebbling graph . . . 27
2.5 A rewrite system, HP, for Horn Pebbling . . . 28
2.6 A rewrite system, CBCP, for Clausal BCP . . . 28
2.7 A rewrite system for CNF transformation . . . 30
2.8 Simplification rules . . . 32
2.9 Propagation rules . . . 33
2.10 Lifting rules . . . 36
2.11 Factoring rules . . . 37
2.12 Rewrite system FP . . . 39
2.13 Equality rules . . . 46
2.14 Rewrite system FPE used for fact propagation with equality . . . 47

3.1 Nesting of procedures in algorithm AFP . . . 51
3.2 Alternative Propagation Rules . . . 52
3.3 Tree representation of $[\![P, ((\neg P \wedge Q) \vee P)]\!]$ . . . 55
3.4 Occurs lists for the tree of $[\![P, ((\neg P \wedge Q) \vee P)]\!]$ . . . 56
3.5 Collapse(N) . . . 65
3.6 Lift1(N) . . . 74
3.7 Lift2(N) . . . 74
3.8 Back pointers for occurs lists . . . 77
3.9 Glits lists . . . 79
3.10 Factoring . . . 83
3.11 Initialization of Occurs arrays of the new nodes created by factoring . . . 86
3.12 Increase in depth due to factoring . . . 88
3.13 ChangeOccurs(M) . . . 92

4.1 Algorithm to compute lfp($T_{R,\Gamma,k}$) . . . 111

5.1 An Algorithm to determine satisfiability . . . 118
5.2 A constraint network . . . 123
5.3 The simplified network . . . 123
5.4 Induced Network . . . 127
5.5 A functional constraint network . . . 134
5.6 OR-database D_0 and query Φ . . . 142
5.7 Tr($D_0, \Phi, ()$) . . . 142
5.8 Testing membership in DLP(k) . . . 149

Abstract

This document addresses some problems raised by the well-known intractability of deductive reasoning in even moderately expressive knowledge representation systems.

Starting from boolean constraint propagation (BCP), a previously known linear-time incomplete reasoner for clausal propositional theories, we develop *fact propagation* (FP) to deal with non-clausal theories, after motivating the need for such an extension. FP is specified using a confluent rewriting systems, for which we present an algorithm that has quadratic-time complexity in general, but is still linear-time for clausal theories. FP is the only known tractable extension of BCP to non-clausal theories; we prove that it performs strictly more inferences than CNF-BCP, a previously-proposed extension of BCP to non-clausal theories.

We generalize a refutation reasoner based on FP to a family of sound and tractable reasoners that are "increasingly complete" for propositional theories. These can be used for anytime reasoning, i.e., they provide partial answers even if they are stopped prematurely, and the "completeness" of the answer improves with the time used in computing it. A fixpoint construction based on FP gives an alternate characterization of the reasoners in this family, and is used to define a transformation of arbitrary theories into logically-equivalent "vivid" theories — ones for which our FP algorithm is complete.

Our final contribution is to the description of tractable classes of reasoning problems. Based on FP, we develop a new property, called bounded intricacy, which is shared by a variety of tractable classes that were previously presented, for example, in the areas of propositional satisfiability, constraint satisfaction, and OR-databases. Although proving bounded intricacy for these classes requires domain-specific techniques (which are based on the original tractability proofs), bounded intricacy is one more tool available for showing that a family of problems arising in some application is tractable. As we demonstrate in the case of constraint satisfaction and disjunctive logic programs, bounded intricacy can also be used to uncover new tractable classes.

Chapter 1

Introduction

1.1 Overview

A fundamental principle underlying much of the research in Artificial Intelligence (AI) is that any intelligent activity requires an explicit body of knowledge related to the task at hand. Such activities include natural language understanding, problem solving, planning, and scene analysis. The field of Knowledge Representation and Reasoning (KR) deals with building and studying systems, known as KR systems, that represent knowledge and draw conclusions from it.

KR systems are often used as embedded utilities or subsystems providing computational services. Levesque [Lev84a] has characterized a KR system as an abstract server supporting two kinds of operations: **Tell**ing the system some additional information (in some restricted formal language $\mathcal{L}_{Tell}$), and **Ask**ing it queries (in some, possibly different, language $\mathcal{L}_{Ask}$). Unlike simple file systems, KR systems can use not just explicitly told information, but also inferred information, in answering questions. For example, if we tell the information "$huge(Mike)$", then the query "$(\exists x)Huge(x) \vee Hungry(x)$" should be answered "true" in most reasonable KR systems, including relational databases.

This relationship between the information told to the system and the answers to queries is usually specified by an entailment (logical consequence) relation in some formal logic. Thus, Tell and Ask operations implement some form of deductive reasoning. The logics most commonly used are first-order logic and propositional logic [Men64, Yas94], since $\mathcal{L}_{Tell}$ and $\mathcal{L}_{Ask}$ are usually subsets of their underlying languages.

It has been cogently argued [BFL83] that for the computational services of a KR system to be dependable, their worst-case time requirements should be small enough to allow adequate response in all critical situations. In other words, Tell and Ask operations should be tractable, which is usually taken to mean that they be in PTIME [GJ79, Yas94]. On the other hand, the language(s) of KR systems should be expressive enough to represent the rich variety of knowledge used in any intelligent activity [DP91]. Not unexpectedly, there is a tradeoff between the expressiveness of a KR system and the tractability of its services — increasing the expressiveness generally decreases the tractability [LB85]. Studying ways to make this tradeoff between expressiveness and tractability, also known as the *intractability problem*, is a central focus of research in KR.

There are several general approaches to the intractability problem (c.f. [Cra92]):

1. Restrict the expressiveness of the KR system (i.e., the language for telling it information and/or the language of asking queries) so that the two operations are provably tractable. Typical examples of this approach include the use of the Horn subset of propositional logic, which is tractable [DG84], and work in relational databases [Ull88], where $\mathcal{L}_{Tell}$ is often just ground facts, but $\mathcal{L}_{Ask}$ is function-free first-order logic. The problem with this approach is that such KR systems are generally too weak to deal with most applications [DP91].
2. Provide a fast, but incomplete (or possibly even unsound), reasoner for the KR system. An incomplete reasoner may fail to infer some information warranted by the underlying logic, while an unsound

reasoner may infer information that is not warranted by the underlying logic. Typical examples of this approach include Boolean Constraint Propagation (BCP) [McA80, McA90] (a variant of unit resolution [CL73]) and tautological entailment [Bel77, Lev84b, Fri87], both of which are sound but incomplete. The general difficulty with this approach is in characterizing (preferably syntactically) the class of queries that will be answered correctly, or the degree of error in the possibly-incorrect answer.

3. An extension to the incomplete/unsound approach employs a *family* of incomplete/unsound reasoners, which differ in their correctness and complexity of reasoning. For any given task, an appropriate reasoner is selected, based on the specific requirements of the task. Typical examples of this approach include the hierarchy of satisfiability problems [GS88] and the family of approximate entailments [CS92a]. In addition to the difficulty in characterizing queries that will be answered correctly by the various reasoners in the family, this approach presents issues concerning the reuse of earlier computation when different reasoners from the family are used.

4. A variant of the incomplete/unsound approach is to *explicitly approximate* the knowledge told to the KB, and/or the queries asked of it, into some other language for which the reasoning is tractable. As in the work on null values in databases [IL82], domain abstraction [Imi87], and knowledge compilation [SK91, BE89], the idea in this approach is to "bound" the error by reporting possibly more than one answer. Note that this approach relies on finding tractable languages to which the formulas are approximated.

This work makes contributions to the first three approaches to the intractability problem.

We start with an widely used exemplar of the second approach: (Clausal) BCP is an efficient (linear-time) reasoning method (i.e., reasoner) for answering clausal propositional queries posed to a KR system that is told clausal propositional information, a problem which is known to be NP-Complete [GJ79, Yas94]. BCP is sound (i.e., logically correct) and incomplete (i.e., does not answer all queries). However, none of its previously-proposed extensions to the non-clausal case are known to be tractable (i.e., provably in PTIME). There are many applications where reasoning with non-clausal theories is required, for example, in verifying automatically generated update constraints [GTT92] and in the applications of logical "Truth Maintenance Systems" (TMS) [de 90, McA90, Mar91]. We develop fact propagation (FP), which tractably extends BCP to non-clausal theories. We present a quadratic-time algorithm for FP, which runs in linear time for clausal theories. Moreover, FP is proved to be more complete than CNF-BCP, a previously-proposed extension of BCP to non-clausal theories. We know of no other reasoner for arbitrary propositional theories, which is tractable and at least as complete as FP.

A second contribution of this work is generalizing a refutation reasoner based on FP to a family of increasingly-complete, sound, and tractable reasoners (the third approach to intractability). Since we show that each theory has a complete reasoner in the family, it can be used for specifying the partial answers of an "anytime reasoner". Anytime reasoners [BD88] are complete reasoners that provide partial answers even if they are stopped prematurely; the completeness of the answer improves with the time used in computing the answer. They could also be used for providing a quick "first cut" to a problem, which can be later improved. Although families of increasingly-complete tractable reasoners were previously-known for the clausal case (c.f. [GS88, CS92a]), we do not know of any other such family of reasoners for arbitrary propositional theories. Our technique for generating these reasoners is based on restricting the length of the clauses used in chaining (i.e., Modus Ponens).

We provide an alternative characterization, based on a fixpoint construction using FP, of the reasoners in our anytime family. This fixpoint characterization is then used to define a transformation of arbitrary propositional theories into logically equivalent theories for which the tractable reasoner FP is complete —- what we will call *"vivid theories"*. Developing appropriate notions of vividness and techniques for compiling theories into vivid theories has already generated considerable interest in the KR community (c.f. [Lev86, Dav91]).

Our final contribution is to the description of tractable cases of reasoning (the first approach to intractability). Based on FP, we develop a new property, called *"bounded intricacy"*, which is shared by a variety of tractable classes that were previously known in the literature, for example, in the areas of (clausal) propositional satisfiability, constraint satisfaction, and so-called "OR-databases". Although proving bounded

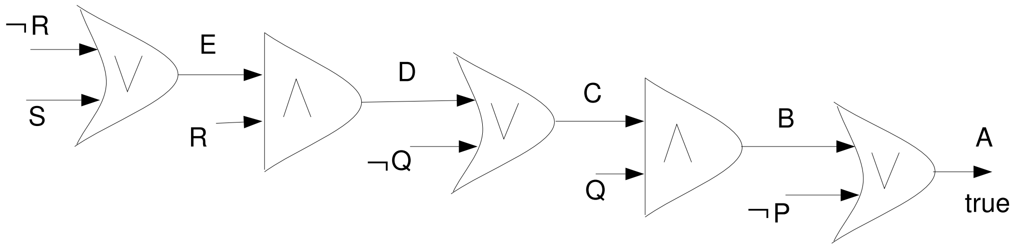

Figure 1.1: Comparing FP and CNF-BCP

intricacy for these classes requires domain-specific techniques, which are based on the original tractability proofs, bounded intricacy is one more tool available for showing that a family of problems arising in some application is tractable. As we demonstrate in the case of constraint satisfaction problems and disjunctive logic programs, bounded intricacy (for low values of intricacy) can be also used to uncover new tractable classes, which can then be checked for applicability. Since there are tractable classes that do not have bounded intricacy, bounded intricacy also seems to provide some new insights into the structure of tractable problems. Filtering out classes with unbounded intricacy may be used as a "first cut" in eliminating intractable classes of reasoning problems.

1.2 Examples

We present three examples that motivate and illustrate our approach to dealing with the intractability problem. In the first example, we compare FP with CNF-BCP for reasoning with a simple digital circuit. In the other examples, we illustrate FP, the family of reasoners, and intricacy using constraint satisfaction problems.

Consider the simple digital circuit of Figure 1.1, where the gates marked $\wedge$ are AND gates and the gates marked $\vee$ are OR gates. A not atypical task in circuit fault diagnosis is to ask what else can be inferred given that both the output A and the input P are *true*.

The formula that encodes the circuit is given by:

$$\psi = \neg P \vee (Q \wedge (\neg Q \vee (R \wedge (\neg R \vee S)))).$$

CNF transformation of ψ produces the following logically equivalent formula:

$$\psi' = (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R \vee S).$$

If we are given the additional observation that P is true, BCP transforms ψ' as follows:

1. infer Q from P and $\neg P \vee Q$
2. infer R from P, Q, and $\neg P \vee \neg Q \vee R$
3. infer S from P, Q, R, and $\neg P \vee \neg Q \vee \neg R \vee S$.

In contrast, FP transforms ψ using P as follows:

1. since both A and P are true, B is also true
2. since B is true, both C and Q are also true
3. since both C and Q are true, D is also true

4. since D is true, both E and R are also true

5. since both R and E are true, S is also true

Note that both FP and CNF-BCP produce the same results.

If this reasoning is to be explained to a novice, whose only knowledge of logic is the truth tables of AND and OR gates, the steps of FP themselves would be sufficiently clear. However, in the case of CNF-BCP, we should first explain the clauses that are obtained by the CNF transformation, and then explain the steps of BCP. This explanation would be something along the following lines:

1. Since A is true, either B or $\neg P$ is true.

2. If B is true then both C and Q are true.

   (a) Thus, either Q or $\neg P$ is true (this is how the first clause of ψ' is explained). Since $\neg P$ is false, it follows that Q is true.

   (b) If C is true then either D or $\neg Q$ is true.

   (c) If D is true then both E and R are true.

      i. Thus, either $\neg P$, $\neg Q$, or R is true (second clause of ψ'). Since P and Q are true, it follows that R is also true.

      ii. If E is true then either $\neg R$ or S is true.

      iii. Thus, either $\neg P$, $\neg Q$, $\neg R$ or S is true (third clause of ψ'). Since P, Q, and R are true, it follows that S is also true.

This explanation, based on CNF-BCP, is clearly more complex than the previous explanation, which is based on FP. Moreover, the explanations based on CNF-BCP become more complex for longer circuits of similar kind, while the explanations based on FP remain simple.

Our remaining examples concern constraint satisfaction problems (CSPs). CSPs deal with assigning values to variables so that some given constraints are satisfied. Many important problems in AI and Computer Science can be viewed as special cases of constraint satisfaction — for example, map-coloring, scheduling, temporal reasoning, circuit design, and diagnostic reasoning (see [Kum92] for references and more examples). A CSP [Mac77, Fre78] is specified by a finite set of "variables" and a set of constraints on subsets of these variables. A CSP is said to be consistent iff there is an assignment of a value to each variable such that all the constraints are satisfied; such an assignment is called a solution of the CSP. Determining consistency is known to be CoNP-Complete [Fre78] even for constraint networks, a restricted class of CSPs in which all the constraints are either unary or binary, and are explicitly provided as sets of tuples. Constraint networks are often represented by a graph whose nodes represent variables and unary constraints, and whose arcs represent binary constraints.

In the network C_1 given in Figure 1.2 (a), for example, the value of the variable w could be either a or b, and variables w and z can together take values a and j, a and k, or b and j. If we use atom $u{:}v$ to stand for "*variable u has value v*", then one can translate the constraints of a CSP into formulas of propositional logic, so that the resulting theory is consistent iff the original CSP is consistent. Starting with the translation of C_1, FP effectively removes b from the constraint on variable w and the constraint (b, j) from the edge (w, z), and obtains the network given in Figure 1.2 (b).

Now consider the network C_2 of Figure 1.3 (a), which is obtained by adding a new variable y and changing some constraints in the network C_1 of Figure 1.2 (a). If we explicitly assign value c to the variable x, then FP obtains $\mathbf{f}$ from the new translated theory, thus, inferring that the resulting network is inconsistent. From this, we obtain that:

$$\Gamma \vdash_0 \neg x{:}c$$

where Γ is the translation of the network C_2, and $\vdash_0$ is the weakest member in the family of reasoners based on FP. Based on this, we disallow the value c for the variable x as part of the transformation of Γ that makes the result "more vivid".

(a) Network C_1 (b) FP on C_1

Figure 1.2: FP on a constraint network

(a) Network C_2 (b) Vivid form of C_2

Figure 1.3: Vivid transformation and Intricacy

After disallowing value c for variable x, if we assign value j to variable z then FP again obtains $\mathbf{f}$ from the new translated theory. From this, we obtain that:

$$\Gamma \vdash_1 \neg z{:}j$$

where $\vdash_1$ is the second weakest member in the family of reasoners based on FP. Intuitively, the subscript 1 indicates that only clauses of size at most 1 have been added to Γ before calling FP. Based on this, we also disallow the value j for variable z as part of the vivid transformation of Γ. The resulting theory, obtained by FP after disallowing the values c and j for the variables x and z respectively, corresponds to the network given in Figure 1.3 (b). This theory, which is obtained by the vivid transformation of the theory Γ, is logically equivalent to Γ and is vivid, since FP is complete for reasoning with it. Alternatively, $\vdash_1$ can be said to be complete for reasoning with the original Γ.

Note that if we assign value j to variable z in the original network C_2, then FP does not obtain $\mathbf{f}$ from the translated theory. It follows that:

$$\Gamma \not\vdash_0 \neg z{:}j$$

which illustrates that $\vdash_1$ can make more inferences than $\vdash_0$. Since $\vdash_1$ is the weakest member in the family of reasoners which is complete for reasoning with Γ, the intricacy of Γ is 1. In contrast, the intricacy of the translation of the network given in Figure 1.3 (b) is 0.

1.3 Plan

Chapter 2 presents FP, which extends BCP to non-clausal theories. FP detects more unsatisfiable theories and infers more information for some theories than CNF-BCP, which is BCP applied after clausal transformation. FP is also useful for transforming a theory into a logically equivalent theory which may be "syntactically simpler". FP is *specified* using a term rewriting system, containing four kinds of rules: simplification, which eliminates truth constants, propagation, which replaces propositional symbols by logical constants, lifting, which moves literals out of nested connectives, and factoring, which identifies common literals in subformulas. Using a term rewriting system has the advantage that it is easier to specify global changes and that old formulas that are not needed are automatically discarded. This rewrite system was also a very useful starting point for developing a tractable algorithm. We prove that the rewrite system for FP is convergent — ensuring that each theory can be finitely rewritten to a unique irreducible form — and modular — ensuring that parts of a theory are independently rewritable before rewriting the entire theory. Some alternative rewrite systems are discarded since they do not satisfy these useful properties.

Chapter 3 presents AFP, an algorithm for computing the irreducible form of a finite theory under FP. The algorithm runs in time quadratic in the size of the input theory, and is based upon efficiently locating the sites for rule applications in a tree representation of the theory, using several additional data structures. We identify invariants for the data structures as a way of developing, and arguing the correctness of, the algorithm. If we restrict theories to be clausal, the complexity of AFP is the same as that of (clausal) BCP: linear time.

Chapter 4 presents the family $\vdash_k$ of reasoners for inferring clauses from propositional theories. We first define $\vdash_{FP}$, a refutation reasoner based on FP. We identify some restricted cases in which it is complete, for example, for reasoning with Horn clauses. Any theory for which $\vdash_{FP}$ is complete for inferring clauses is called vivid, a term inspired by [Lev86], where vivid theories are ones where an answer can be "read off" quickly. By adding some inference rules, which allow chaining on previously inferred formulas, we extend $\vdash_{FP}$ to a sound and complete reasoner. By restricting this chaining, we obtain the family $\vdash_k$ of reasoners: for any number k, the reasoner $\vdash_k$ allows chaining over clauses of size at most k. We also present a function Viv, defined in terms of lattice-theoretic fixed-points, such that for every Γ there is a k for which $\mathrm{Viv}(\Gamma, k)$ is vivid. We show that the set of clauses inferable from $\mathrm{Viv}(\Gamma, k)$ using $\vdash_{FP}$ is exactly the set of clauses inferable from Γ using $\vdash_k$. Since our results do not depend on the exact details of FP, we abstract out those properties of rewrite systems that are required, thus providing some degree of generality.

Chapter 5 presents our technique for describing tractable cases of satisfiability based on the notion of intricacy: for any theory Γ, the least value k for which $\mathrm{Viv}(\Gamma, k)$ is vivid is said to be the *intricacy* of Γ.

We show that satisfiability is tractable for any class of theories such that all the *unsatisfiable* theories in the class have intricacy at most k, for some fixed constant k. Although this "bounded intricacy" criterion is a sufficient condition for tractability, we show that there are tractable classes that do not have bounded intricacy (so the notion of "bounded intricacy" is not synonymous with "tractable"). We then show that some tractable classes already presented in the literature do have bounded intricacy. These include tractable classes of OR-databases [IMV94] and CSPs [DP88, DH91]. We also describe some new tractable classes using the bounded intricacy criterion. These include the first non-obvious tractable class in disjunctive logic programs [LJR92], and a new tractable family of classes in CSP.

Chapter 6 reviews the contributions of this thesis and presents some useful directions for future research.

Chapter 2

Fact Propagation

2.1 Overview

Our main goal is to develop a tractable extension of Boolean Constraint Propagation (BCP) to knowledge bases represented by non-clausal theories. For clausal theories, this extension should have the same time complexity as BCP.

BCP [McA80, McA90] is an efficient (linear time) but incomplete method for reasoning with finite clausal theories in propositional logic. For example, BCP transforms the theory $\{P, \neg Q, (\neg P \vee Q \vee R)\}$ to $\{P, \neg Q, R\}$. In applications like truth-maintenance systems, BCP is used for inferring literals (i.e., facts) entailed by a given theory.

Some applications require reasoning with non-clausal theories. Two such examples are automatically generated database update constraints and the propositional translation of a constraint satisfaction problem. However, the standard extensions of BCP to non-clausal theories — Formula-BCP, Prime-BCP and CNF-BCP — do not have known tractable algorithms. In addition, conjunctive normal form transformation of a theory, the first step in CNF-BCP, has the disadvantage that the formulas in the theory lose their natural form, so explanations become difficult.

We present fact propagation (FP), a technique that detects more unsatisfiable theories and infers more facts for some theories than CNF-BCP. FP is also useful for transforming a theory into a logically equivalent theory which may be "syntactically simpler".[1] For example, FP simplifies the theory $\{(P \vee (Q \wedge (\neg Q \vee P)))\}$ to $\{P\}$, while CNF-BCP stops after transforming it to $\{(P \vee Q), (P \vee \neg Q)\}$. We observe that BCP on clausal theories can be viewed as a two step process — propagate and simplify: propagation replaces propositional symbols by logical constants; e.g., $\{(P \vee (Q \wedge (\neg Q \vee P)))\}$ becomes $\{(P \vee (Q \wedge (\neg \mathbf{t} \vee \mathbf{f})))\}$; simplification reduces this to P. FP generalizes these steps to non-clausal theories, and adds some new steps. FP itself is incomplete; for example, it does not infer P from the theory $\{(P \vee Q), (P \vee \neg Q), (Q \vee R)\}$

FP is *specified* in terms of a rewrite system. This has the advantage that it is easier to specify global changes (as in propagation) and that old formulas that are not needed are automatically discarded (e.g., after simplification). We were not able to specify FP using the standard techniques used in logic, namely model theory or inference rules. This rewrite system was also a very useful starting point for developing a tractable algorithm.

We prove that the rewrite system FP is confluent, terminating, and modular. Confluence, which ensures that the final result does not depend on the particular ordering of rule applications, provides us a welcome degree of freedom in developing the tractable deterministic algorithm and arguing for its correctness. Termination, which ensures that every sequence of rule applications terminates, allows us to prove confluence using the Knuth-Bendix approach [KB70] and helps us in developing the tractable algorithm. Modularity, which ensures that parts of a theory are independently rewritable before rewriting the entire theory, is useful

[1] The actual measure of simplicity is formally presented in Definition 2.18.

for knowledge bases that are built incrementally. Some alternative rewrite systems are discarded since they do not satisfy these useful properties.

In addition to propagation and simplification, FP has two additional kinds of rules: lifting, which moves literals out of nested connectives, and factoring, which identifies common literals in subformulas. For example, one kind of lifting rewrites $(P \wedge (Q \wedge (R \vee S) \wedge (R \vee T)))$ to $(P \wedge Q \wedge ((R \vee S) \wedge (R \vee T)))$; in turn, this rewrites to $(P \wedge Q \wedge R \wedge (S \vee T))$ by using factoring and some simplification. These rules are needed to make FP infer more facts than CNF-BCP, and to keep it confluent.

2.2 Mathematical Preliminaries

We normally use $\mathcal{N}$ to denote the set $\{0, 1, 2, \ldots\}$ of natural numbers and n to denote any element of this set. For any n, the set $\{0, \ldots, n\}$ contains at least the number 0, while the set $\{1, \ldots, n\}$ is empty when $n = 0$. The empty set is denoted by $\emptyset$.

For any *binary relation* $\rightarrow$, we use $\leftarrow$, $\leftrightarrow$, $\rightarrow^+$, and $\rightarrow^*$ to denote its inverse, symmetric closure, transitive closure, and reflexive-transitive closure, respectively. Thus, $\leftrightarrow^*$ denotes the equivalence closure of $\rightarrow$.

A *bag* (also called *multiset*), denoted by $[\![\ldots]\!]$, is a **finite** collection of elements in which elements can occur more than once. Intuitively, it is a finite set in which multiplicity of elements is considered significant. A bag can also be viewed as a function which assigns a number to each element, denoting the number of occurrences of the element in the bag. The empty bag is denoted by $[\![]\!]$. $x \in B$ denotes that x is an element of bag B. Bag A is a *subbag* of bag B, denoted by $A \subseteq B$, iff each occurrence of an element in A has a distinct corresponding occurrence in B. The *union* of two bags A and B, denoted by $A \cup B$, is the bag in which the number of occurrences of any element is the sum of number of occurrences of that element in A and B. The *intersection* of two bags A and B, denoted by $A \cap B$, is the bag in which the number of occurrences of any element is the minimum of number of occurrences of that element in A and B.

A *tuple*, denoted by $\langle \ldots \rangle$ is a sequence of elements. Intuitively, it is a bag in which the ordering of elements is considered significant. For any tuples r and t, the tuple $r \diamond t$ is obtained by appending t to the right of r.

A *well-ordering* on a set is any partial (irreflexive) ordering with no infinite decreasing chains.

The *cardinality* $|S|$ of any set, bag, or tuple S is the number of elements (including duplicates) in S.

2.3 PCE : Propositional Calculus with Equality and Generalized Connectives

We present a variant, PCE, of propositional calculus, PC [Men64], which has equality as well as generalized conjunction and disjunction. Generalized connectives, which allow any number of arguments [Fit90], are used because facts inferred using FP will depend on the grouping of the formulas, which are always in negation normal form. Bags, instead of sets, are used as arguments of these connectives and for constructing theories, which are always finite, since we do not want our algorithms to have to detect and eliminate multiple occurrences of formulas. We define a simple truth functional semantics of PCE by extending that of PC to deal with equality. We also define a notion of facts that are directly inferable from a theory, i.e., without using complicated reasoning steps.

2.3.1 Syntax

We assume that we have a denumerable set $\mathcal{P}$ of symbols called *predicates*, each of which has a number $n \in \mathcal{N}$ associated with it. If number n is associated with predicate P, we say P is an n-place predicate; n is also called the *arity* of P. Set $\mathcal{P}$ is required to contain a special 2-place predicate $\doteq$ called *equality*. We also assume that we have a denumerable set $\mathcal{C}$ of symbols called *individual constants* (or just constants), along with some total well-ordering $\succ$ among the elements of $\mathcal{C}$ (i.e., there is no infinite decreasing sequence $a_1 \succ a_2 \succ \ldots$ of constants in $\mathcal{C}$). We require that sets $\mathcal{P}$ and $\mathcal{C}$ be disjoint and not contain the two special symbols, $\mathbf{t}$ and $\mathbf{f}$, called *logical constants*. We normally use symbols P, Q, etc. to denote predicates, and symbols a, b, etc. to denote individual constants.

The atoms of PCE are the counterpart of propositions of PC. They are built from predicates and constants. Literals are defined as usual.

Definition 2.1 An *atom*, p, is an expression of the form $P(a_1, \ldots, a_n)$ where $n \in \mathcal{N}$, P is a n-place predicate in $\mathcal{P}$, and $a_1, \ldots, a_n$ are constants in $\mathcal{C}$. Each a_i $(i \in 1 \ldots n)$ is called an *argument* of the atom p, and P is called the *predicate* of p. If n=0 then the atom $P()$ is abbreviated as P and is called a *proposition*. A *literal*, α, is either an atom p, or its negation $\neg p$; atom p is called the *atom* of α. ∎

Atoms are also called *positive* literals, while their negations are called *negative* literals. Some examples of atoms are R, $P_1(a, b, c)$, $Q(a, a)$, and $\doteq(a, c)$, where R is a 0-place predicate, P_1 is a 3-place predicate, Q is a 2-place predicate, and a, b and c are constants. We normally use symbols p, q, etc. to denote atoms, and symbols α, β, etc. to denote literals.

Note: The logical constants $\mathbf{t}$ and $\mathbf{f}$ are **not** considered to be literals.

Formulas are usually built from atoms using some fixed set of connectives. Our formulas are constructed from literals using only the connectives *conjunction* ($\wedge$) and *disjunction* ($\vee$). Following [Fit90], we allow any number of arguments for these connectives. This allows more flexibility in specifying formulas, since facts inferred from a formula using FP will depend on how its components are grouped with respect to these connectives. For technical convenience, all the arguments of a connective are grouped together in a single bag.

Definition 2.2 *Formulas* are defined inductively as follows:

1. any literal α is a formula;
2. if B is a bag of formulas, then $\wedge(B)$ and $\vee(B)$ are formulas.

The connectives $\wedge$ and $\vee$ are called *formula connectives*. The formula $\wedge(B)$ is a called a *conjunctive formula* and each formula in B is called a *conjunct*. $\vee(B)$ is a called a *disjunctive formula* and each formula in B is called a *disjunct*. The formulas $\wedge(\llbracket\rrbracket)$ and $\vee(\llbracket\rrbracket)$ are abbreviated by the special symbols $\mathbf{t}$ and $\mathbf{f}$, respectively. ∎

We use bags as arguments instead of plain sets since we do not want to detect and eliminate multiple occurrences of formulas, an expensive operation, in our algorithms. Moreover, as we shall see later, retaining multiple occurrences of formulas does not effect our results. Since negation symbols appear only in front of atoms, formulas constructed in this manner are usually said to be in *negation normal form* (NNF). Note that $\wedge(\llbracket\rrbracket)$ and $\vee(\llbracket\rrbracket)$ are also logical constants. We normally use symbols ψ, φ, etc. to denote formulas.

Note: Negation, $\neg$, is **not** considered to be a connective.

Some examples of formulas are given below:

$$
\begin{gathered}
P \\
\neg \doteq(a, b) \\
\wedge(\llbracket P, \vee(\llbracket \neg P, Q \rrbracket), \wedge(\llbracket\rrbracket)\rrbracket) \\
\wedge(\llbracket P, \vee(\llbracket \neg P, Q \rrbracket), \mathbf{t}\rrbracket) \\
\vee(\llbracket \wedge(\llbracket P, \doteq(a, b)\rrbracket), R(a, c), R(b, c)\rrbracket)
\end{gathered}
$$

where P and Q are 0-place predicates, R is a 2-place predicate, and a, b, and c are constants. The first two formulas are literals, the third and fourth denote the same conjunctive formula with three conjuncts, and the last is a disjunctive formula with three disjuncts.

Notational conventions: For the sake of readability, we normally omit the bag constructor $[\![\ldots]\!]$ from the argument of the connectives. For example, the formula $\vee([\![\wedge([\![P, a \doteq b]\!]), R(a,c), R(b,c)]\!])$ is rendered as $\vee(\wedge(P, a \doteq b), R(a,c), R(b,c))$. Moreover, when using typed expressions (e.g., B, B_i have type "bag of formula", while $\varphi, \psi, \ldots$ have type "formula") we will overload the comma operator so that the following pairs are considered equivalent:

$$
\begin{array}{lll}
\wedge(\psi, B) & \text{and} & \wedge([\![\psi]\!] \cup B) \\
\wedge(B, \psi) & \text{and} & \wedge(B \cup [\![\psi]\!]) \\
\wedge(B_1, B_2) & \text{and} & \wedge(B_1 \cup B_2) \\
\wedge(\psi_1, \ldots, \psi_n) & \text{and} & \wedge([\![\psi_1, \ldots, \psi_n]\!])
\end{array}
$$

Also, the special predicate $\doteq$ is treated as an infix operator: for any constants a and b, the atom $\doteq(a,b)$ and the literal $\neg \doteq(a,b)$ are written as $a \doteq b$ and $a \not\doteq b$, respectively.

We need the notion of the complement of a formula. Since we keep formulas in negation normal form, taking their complement requires moving the outer negation all the way inside to the front of atoms.

Definition 2.3 The *complement*, $\sim\psi$, of a formula ψ is defined inductively as follows:

1. if p is an atom then $\sim p = \neg p$ and $\sim\neg p = p$;
2. $\sim\wedge(\psi_1, \ldots, \psi_n) = \vee(\sim\psi_1, \ldots, \sim\psi_n)$;
3. $\sim\vee(\psi_1, \ldots, \psi_n) = \wedge(\sim\psi_1, \ldots, \sim\psi_n)$

■

Some examples of complement are:

$$
\sim\wedge(P, \vee(\neg P, Q), \neg S) = \vee(\neg P, \wedge(P, \neg Q), S)
$$
$$
\sim\vee(\wedge(P, a \doteq b), Q(a,c), \neg Q(b,c)) = \wedge(\vee(\neg P, a \not\doteq b), \neg Q(a,c), Q(b,c))
$$

As special cases, we get $\sim\mathbf{t} = \mathbf{f}$ and $\sim\mathbf{f} = \mathbf{t}$.

Theories are normally collections of formulas. Our theories are built from bags of formulas and the connective $\odot$, which serves to distinguish theories from bags that are arguments of $\wedge$ and $\vee$:

Definition 2.4 A theory is an expression of the form $\odot(B)$ where B is a bag of formulas. The connective $\odot$ is called the *theory connective*. ■

Since bags are finite, our theories are also finite. We shall see later that the rewrite system FP treats a theory differently from a conjunction built from the same arguments, although both are considered to be terms that are rewritable. Thus, our syntax for theories is similar, but not identical, to conjunctive formulas. Like formulas, theories constructed in this manner are also said to be in negation normal form. We normally use symbols Γ, Δ, etc. to denote theories.

We also define some special formulas and theories in the usual way:

Definition 2.5 A *clause* is a formula $\vee(B)$ where B is a bag of literals or logical constants. A *Horn clause* is a clause $\vee(B)$ where B has at most one positive literal. A *clausal theory* is a theory $\odot(B)$ where B is a bag of clauses. A *Horn theory* is a theory $\odot(B)$ where B is a bag of Horn clauses. For any number k, a *k-CNF theory* is a clausal theory in which each clause has at most k literals. A *positive theory* is a theory in which all the literals are positive. A *negative theory* is a theory in which all the literals are negative. ■

A clausal theory is also said to be in *conjunctive normal form* (CNF). We will also use notions of subtheory and subclause that refer to parts of theories and clauses, respectively:

Definition 2.6 Any subbag of a theory is called its *subtheory*. Clause $\vee(B)$ is a *subclause* of clause $\vee(B')$ iff $B \subseteq B'$. Clause ψ is a *proper subclause* of clause φ iff ψ is a subclause of φ and φ is not a subclause of ψ. Clause ψ is an *immediate subclause* of clause φ iff ψ is a proper subclause of φ and there is no clause μ such that ψ is a proper subclause of μ and μ is a proper subclause of φ. ■

2.3.2 Semantics

A model-theoretic semantics for the propositional language PCE is obtained in the usual way by mapping the atoms to the truth values *true* and *false* [Men64], with the additional requirement that each mapping be consistent with equality. Thus, we avoid the additional machinery of defining domains and mapping the constants of PCE to objects in a domain.

Definition 2.7 An interpretation is any mapping v from atoms to the set $\{true, false\}$ of truth values such that:

1. $v(a \doteq a) = true$ for any constant a;
2. for any n, any n-place predicate P (including $\doteq$), and any constants $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$: if $v(a_i \doteq b_i) = true$ for each $i = 1 \ldots n$ and $v(P(a_1, \ldots, a_n)) = true$ then $v(P(b_1, \ldots, b_n)) = true$.

An interpretation is often compactly specified by the subset of atoms that are mapped to *true*.

Any interpretation v on atoms can be extended to all formulas and theories as follows:

1. for any atom p, $v(\neg p) = true$ iff $v(p) = false$;
2. for any bag B of formulas, $v(\wedge(B)) = v(\odot(B)) = true$ iff $v(\psi) = true$ for each formula ψ in B;
3. for any bag B of formulas, $v(\vee(B)) = true$ iff $v(\psi) = true$ for some formula ψ in B.

■

It follows that the truth values of conjunctions, disjunctions, and theories do not depend on the ordering of the formulas in their bag argument. Also, $v(\wedge(\llbracket\rrbracket)) = true$ and $v(\vee(\llbracket\rrbracket)) = false$ for any interpretation v; thus, our decision to use the symbols **t** and **f** for $\wedge(\llbracket\rrbracket)$ and $\vee(\llbracket\rrbracket)$, respectively, is semantically justified. The notion of complement is also semantically justified, since for any interpretation v and any formula ψ, $v(\sim\psi) = true$ iff $v(\psi) = false$.

As expected, equality $\doteq$ behaves like an equivalence relation with respect to interpretations, i.e., for any constants a, b, and c and any interpretation v: $v(a \doteq a) = true$, $v(a \doteq b) = v(b \doteq a)$, and if $v(a \doteq b) = v(b \doteq c) = true$ then $v(a \doteq c) = true$. (The latter two identities follow from part (2) of the definition, when P is set to $\doteq$.) We will exploit this property in the syntax by considering the atoms $a \doteq b$ and $b \doteq a$ to be identical, i.e., $a \doteq b \;=\; b \doteq a$ where $=$ is the usual metalevel equality construct that relates identical or equal items.

Notions of satisfiability, model, entailment, and equivalence are defined as usual:

Definition 2.8 A theory, Γ, is *satisfiable* iff there is an interpretation v for which $v(\Gamma) = true$; interpretation v is then called a *model* of Γ. A formula ψ is *logically entailed* (or just *entailed*) by a theory Γ iff $v(\psi) = true$ for all interpretations v such that $v(\Gamma) = true$; we denote this by $\Gamma \models \psi$. Theories (or formulas) Γ and Δ are *logically equivalent*, denoted by $\Gamma \equiv \Delta$, iff $v(\Gamma) = v(\Delta)$ for each interpretation v. ■

2.3.3 Alternative Syntax for Examples

Although we will develop the theory using the above notation, a more standard notation for predicate calculus formulas and theories can be recovered using the following rules:

1. use **t** and **f** for $\wedge(⟦⟧)$ and $\vee(⟦⟧)$ respectively;
2. use (ψ) for both $\wedge(⟦\psi⟧)$ and $\vee(⟦\psi⟧)$;
3. use $(\psi_1 \wedge \ldots \wedge \psi_n)$ for $\wedge(⟦\psi_1, \ldots, \psi_n⟧)$ where $n > 1$;
4. use $(\psi_1 \vee \ldots \vee \psi_n)$ for $\vee(⟦\psi_1, \ldots, \psi_n⟧)$ where $n > 1$;
5. use $⟦\psi_1, \ldots, \psi_n⟧$ for $\odot(⟦\psi_1, \ldots, \psi_n⟧)$;

Note that $\wedge(\psi)$ and $\vee(\psi)$ are semantically equivalent, so we are justified in using the same alternative notation for both of them. However, we will use the original notation for such formulas in cases where we must make a distinction. Also, the context should distinguish whether a bag of formulas is used as a theory or as an argument to a formula connective.

For example, some of the example formulas given earlier can be expressed in this modified syntax as:

$$P$$
$$a \not\doteq b$$
$$((P \wedge a \doteq b) \vee R(a, c) \vee R(b, c))$$

However, the following are **not** formulas in either notation:

$$\begin{array}{ll} P \wedge Q & \text{(no parentheses)} \\ (\neg(P \vee Q)) & \text{(improper negation)} \end{array}$$

Exploiting (5) above, we also extend the usual operations on bags to theories. For example, if $\Gamma = \odot(B)$ and $\Gamma' = \odot(B')$ where B and B' are any bags of formulas, then

- for any formula ψ, $\psi \in \Gamma$ iff $\psi \in B$;
- $\Gamma \cup \Gamma' = \odot(B, B')$;
- etc.

2.3.4 Directly Inferable Facts

We are interested in simple formulas that can be directly inferred from a theory, i.e., inferred without any complicated reasoning steps:

Definition 2.9 Any literal or logical constant (**t** or **f**) is called a *fact*. ∎

Facts are special in several ways. First, each of them force a definite, unambiguous constraint on the interpretations that make the fact true. Second, they are the simplest such formulas; for example, although the formula $\wedge(P, Q)$ unambiguously forces the truth values of both P and Q, it can be simplified to the set $\{P, Q\}$ of facts. Third, facts inferred from a theory can be efficiently used to simplify the theory, as we shall see later in this paper.

In a clausal theory, a fact α (say, some literal) would be represented as the clause $\vee(\alpha)$. Thus, we should be able to infer α from the formula $\vee(\alpha)$. Generalizing this observation, we define the following notion of facts that are directly inferable from a theory or a formula:

Definition 2.10 For any formula ψ, the set $\text{facts}(\psi)$, called the set of facts *directly inferable* from ψ, is defined inductively as follows:

1. $\text{facts}(\mathbf{t}) = \{\mathbf{t}\}$;

2. facts($\mathbf{f}$) = set of all facts;

3. facts(α) = $\{\alpha, \mathbf{t}\}$, for any literal α;

4. facts($\wedge(\psi)$) = facts($\vee(\psi)$) = facts(ψ), for any fact ψ;

5. facts(ψ) = $\{\mathbf{t}\}$, for any other formula ψ.

For any theory $\odot(B)$, the set facts($\odot(B)$), called the set of facts directly inferable from $\odot(B)$ is given by:

$$\text{facts}(\odot(B)) = \bigcup_{\psi \in B} \text{facts}(\psi)$$

■

For example, consider the theory $\Gamma = \odot(\wedge(Q), \wedge(\vee(P)))$. Since facts$(\wedge(Q)) = \{Q, \mathbf{t}\}$ and facts$(\wedge(\vee(P))) = \{\mathbf{t}\}$, we obtain that facts$(\Gamma) = \{Q, \mathbf{t}\}$.

It follows that, for any theory Γ, the truth values of formulas in the set facts(Γ) in any model of Γ must be set to true. These are the facts that can be directly-inferred from the theory, without any further reasoning. Notice that the set degenerates to the set of all facts when $\mathbf{f}$ can be so inferred.

We could have used a more liberal definition, for example, allowing fact P to be inferable from the formula $\wedge(P, P)$. However, we expect reasoning algorithms to simplify this formula to either $\wedge(P)$ or P, from which fact P can be inferred. Also, no fact is directly inferable from the formula $\wedge(\wedge(P))$, because of a technical difficulty this would cause with a later proof (Lemma 2.22).

In the presence of equality, the definition of facts is extended so that many additional facts can be inferred. For example, the fact $a \doteq a$ for any constant a is always inferable, even from an empty theory. Also, if $\neg P(a)$ and $a \doteq b$ are inferable, then $\neg P(b)$ can also be inferred.

Definition 2.11 For any set A of facts, the *equality closure*, $A^{\doteq}$, of A is the smallest set of facts such that:

1. for any constant a, $a \doteq a \in A^{\doteq}$;

2. $A \subseteq A^{\doteq}$;

3. for any n, any n-place predicate P, and any constants $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ such that $\{a_i \doteq b_i \mid i = 1 \ldots n\} \subseteq A^{\doteq}$: if $P(a_1, \ldots, a_n) \in A^{\doteq}$ then $P(b_1, \ldots, b_n) \in A^{\doteq}$, and if $\neg P(a_1, \ldots, a_n) \in A^{\doteq}$ then $\neg P(b_1, \ldots, b_n) \in A^{\doteq}$.

The set of facts *inferable using equality* from a theory Γ is given by the set facts$(\Gamma)^{\doteq}$. ■

It follows directly from the above definition that for any bags A and B of facts, if $A \subseteq B$ then $A^{\doteq} \subseteq B^{\doteq}$. Note that the equality closure does not produce all possible deductions based on equality; for example, $a \not\doteq b$ is not in the equality closure of the set $\{P(a), P(b)\}$.

Note the distinction between "infer" and "entail": entailment relation between theories and formulas is a semantic property of the logic, independent of any specific algorithm, while an inference relation between theories and formulas is specified by a particular algorithm (syntactic method). An inference method is considered to be *sound* iff each formula inferred from any theory is also entailed by it; it is considered to be *complete* iff each formula entailed by a theory can be inferred from it. In this section, we presented a sound and incomplete inference technique. In later sections, we will present other sound inference techniques.

2.3.5 Discussion

Our calculus, PCE, differs from the standard propositional calculus, PC, [Men64] in several ways:

1. PC requires atoms to be only propositions, while PCE allows atoms to be also built from predicates and constants;
2. PCE allows the special equality predicate $\doteq$ in atoms, while PC does not allow this predicate;
3. PC allows theories to be denumerable, while PCE restricts them to be finite.

There are also some syntactic differences. First, PC restricts conjunctions and disjunctions to be binary, while PCE allows them to have any finite number of arguments (recall that these arguments are put together in a single bag of formulas). Second, PCE restricts the formulas to be in negation normal form, while PC allows $\neg$ as a unary connective that can appear in front of any formula (not just atoms).

We will use "finite PC" to denote PC restricted to finite theories. It follows that finite PC is a syntactic variant of PCE without equality.

2.4 Rewrite Systems for PCE

We present a variant notion of rewrite systems [DJ90, KB70, Hue80, Der89] applicable to theories in PCE, in which groups of rewrite rules are represented by rule schemas that contain meta-variables. These rewrite systems will be used to rewrite theories into logically equivalent theories that are syntactically simpler, based on a measure of simplicity defined in this section. As is usual, convergence of a rewrite system ensures that each theory rewrites to a unique irreducible theory using a finite number of rewrite steps. We introduce some additional properties, including modularity and monotonicity, which are desirable since our rewrite systems will be used for logical reasoning with knowledge bases. Since these properties are global to a rewrite system, we will develop some techniques to prove that a rewrite system satisfies them by considering only individual rules or pairs of rule schemas.

2.4.1 Rewrite Systems

A rewrite system, also called a term-rewriting system, specifies rewrite rules for manipulating terms, which are symbolic structures defined inductively using constants, variables, and function symbols (also called functors). Since a rule is applied by replacing a part of a term, called a subterm, by a different subterm, we need a way to specify subterms of a term.

Our terms are constants, formulas, and theories of PCE:

Definition 2.12 Constants in $\mathcal{C}$, formulas, and theories are all terms. They are of three sorts, based on the functors used to build them:

1. any constant in $\mathcal{C}$ is a $\mathcal{C}$-term;
2. formula terms are constructed using n-ary functors P and $\neg P$ for each n-place predicate P in $\mathcal{P}$, and logical connectives $\wedge$ and $\vee$;
3. theory terms are constructed using the functor $\odot$.

Formula and theory terms are also called *logical terms.* ■

We normally use symbols s, t, etc. to denote terms. Usually, terms are viewed as trees, with subtrees being called subterms.

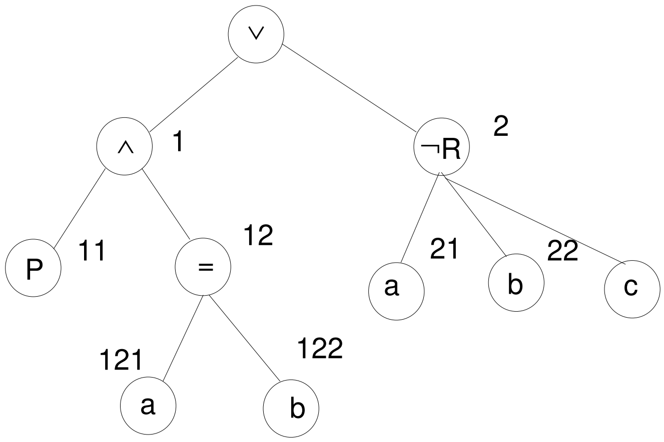

Figure 2.1: Logical term as a tree

Definition 2.13 *Immediate subterms* of terms are defined as follows:

1. each t_i $(i \in 1 \ldots n)$ is an immediate subterm of $P(t_1, \ldots, t_n)$ and of $\neg P(t_1, \ldots, t_n)$.
2. any $t \in B$ is an immediate subterm of $\wedge(B)$, $\vee(B)$, and $\odot(B)$.

For any term t, its *proper subterms* are defined as follows:

1. any immediate subterm of t is a proper subterm of t;
2. any proper subterm of any immediate subterm of t is a proper subterm of t.

Term s is a *subterm* of term t iff either s is the same as t or s is a proper subterm of t. A formula subterm is also called a *subformula*. ∎

Note that the notions of subtheory and subclause, as defined in Section 2.3.1, are a little bit different from the above notion of a subterm. Rather than being subtrees of the tree representing the term, subtheories and subclauses are obtained by removing subtrees at some children of the root. Note also that P is not a subterm of $\neg P$.

It is customary in rewriting systems to identify subterms by the their roots in a notation similar to the Dewey-decimal notation. The term itself is at position Λ, which denotes the empty string. For example, if t is $\vee(\wedge(P, a \doteq b), \neg R(a, b, c))$, shown as a tree in Figure 2.1, then the following are some of its subterms:

$$\begin{array}{rll} t|_\Lambda & is & \vee(\wedge(P, a \doteq b), \neg R(a, b, c)) \\ t|_1 & is & \wedge(P, a \doteq b) \\ t|_{11} & is & P \\ t|_{12} & is & a \doteq b \\ t|_{122} & is & b \\ t|_2 & is & \neg R(a, b, c) \\ t|_{21} & is & a \end{array}$$

There are some operations on terms which will be useful for our rewriting system:

Definition 2.14 [Replacement] For any subterm s of term t and any term r, where s and r are of the *same sort* ($\mathcal{C}$, formula, or theory):

1. the term $t_\pi[r \leftarrow s]$ is obtained from t by replacing subterm s at position π by r.
2. the term $t[r \stackrel{*}{\leftarrow} s]$ is obtained from t by replacing *each* occurrence of subterm s in t by r.
3. if s and r are formula terms, then the term $t[r \stackrel{*\sim}{\leftarrow} s]$ is obtained from t by replacing *each* occurrence of subterm s in t by r and $\sim s$ by $\sim r$.

■

Intuitively, the superscripts $*$ and $\sim$ denote "all occurrences" and "complement also", respectively. For example, if $t = \vee(\wedge(P, a \doteq b, \neg P), \neg R(a,b,c))$ then

$$\begin{aligned}
t_{21}[c \leftarrow a] &= \vee(\wedge(P, a \doteq b, \neg P), \neg R(c,b,c)) \\
t[c \stackrel{*}{\leftarrow} a] &= \vee(\wedge(P, c \doteq b, \neg P), \neg R(c,b,c)) \\
t[P \stackrel{*}{\leftarrow} Q] &= \vee(\wedge(Q, a \doteq b, \neg P), \neg R(a,b,c)) \\
t[P \stackrel{*\sim}{\leftarrow} Q] &= \vee(\wedge(Q, a \doteq b, \neg Q), \neg R(a,b,c))
\end{aligned}$$

Since terms are replaced by terms of the same sort, all replacements in terms necessarily produce well-defined terms. For example, $t[\neg P \stackrel{*\sim}{\leftarrow} a]$ is not allowed since $\neg P$ and a are not of the same sort.

Note also that since the order of elements in a bag is not significant, the order of arguments for the functors representing logical connectives is also not significant when considering term equality $=$. Similar observation holds for the order of arguments of the equality predicate $\doteq$.

A rewrite (or term-rewriting) system is a collection of rewrite rules, where each rewrite rule is a directed pair of logical terms of the same sort. Each rewrite rule specifies how a term can be rewritten:

Definition 2.15 A *rewrite rule* is of the form $l \Rightarrow r$, where l and r are both either theories or formulas; the rule is called a *theory rule* or a *formula rule*, respectively. A *rewrite system* R is a set of rewrite rules. For any rewrite system R, a term s *rewrites* to term t, denoted by $s \Rightarrow_R t$, if there is a rewrite rule $l \Rightarrow r$ in R and a position π in s such that l is a subterm of s at π and $t = s_\pi[r \leftarrow l]$. ■

In fact, we will represent groups of rewrite rules by *rule schemas*, containing meta-variables denoting constants, atoms, literals, formulas, and bags of formulas. Each schema represents all *rule instances* that are obtained by substituting for all its meta-variables by terms of the appropriate type. This assignment of terms for meta variables is called a *substitution*. Occasionally, we will impose conditions on the kinds of terms that can be substituted for some meta-variable (e.g., "bag B cannot be empty"), in which case only those instances of the rule schema satisfying these conditions are to be considered.

For example, if $\vee(\mathbf{f}, B) \Rightarrow \vee(B)$ is a rewrite rule in R, where meta-variable B denotes a bag of formulas, then using its rule instance $\vee(\mathbf{f}, Q, S) \Rightarrow \vee(Q, S)$, obtained using the substitution $\sigma = \{[\![Q, S]\!]/B\}$, we get:

$$\vee(Q, \wedge(P, \vee(\mathbf{f}, Q, S))) \Rightarrow_R \vee(Q, \wedge(p, \vee(Q, S)))$$

Henceforth, we shall usually not deal explicitly with the substitution of meta-variables, treating instead rule schemas as "prototypical" rule instances: for any rule schema $l \Rightarrow r$, our definition of $t_\pi[r \leftarrow l]$ presupposes a particular rule instance such that the instantiation of the left hand side of the rule produces the subterm of t at π. In other words, if $t|_\pi = s$ and $s = l\sigma$ for some substitution σ, then $t_\pi[r \leftarrow l]$ is obtained by replacing s in t by $r\sigma$.

When a rewrite system has a name, that name is used instead of the subscript R in $\Rightarrow_R$. For nameless systems, we continue to use the subscript R to distinguish $\Rightarrow_R$ from $\Rightarrow$. Note also that we are dealing with three different kinds of transformations:: *substitution* of meta-variables in a rule schema produces a rule instance; a term *rewrites* to another using a rewrite rule; a rewrite rule is often expressed using *replacement* of subterms.

We are often interested in terms that cannot be rewritten:

Definition 2.16 A term s is *irreducible* in a rewrite system R if there is no term t such that $s \Rightarrow_R t$. If $s \Rightarrow_R^* t$ and t is irreducible then we say that t is an *R-normal form* of s, or that s *reduces* to t; we denote this by $s \Rightarrow_R^! t$ where $\Rightarrow_R^!$ is called the *reduction relation* induced by the rewrite system R. ∎

2.4.2 Properties of Rewrite Systems

To be suitable for tractable reasoning with knowledge bases, we argue that a rewrite system should satisfy certain properties that are presented in this section. Since there are many possible rewrite systems, we will also use these properties as informal heuristics for developing rewrite systems suitable for our reasoning task. In the discussion below, recall that $\Leftrightarrow_R^*$ and $\Rightarrow_R^*$ denote the equivalence closure and reflexive-transitive closure, respectively, of $\Rightarrow_R$ for any rewrite system R. Since $\Leftrightarrow_R^*$ is an equivalence relation, it partitions the set of all logical terms into equivalence classes: $[s]_R = \{t \mid s \Leftrightarrow_R^* t\}$. The properties of interest to us are:

Confluence: Reasoning using a confluent rewrite system produces the same result for a specific theory, irrespective of the order in which the rewrite rules are applied. This has two important consequences. An algorithm computing the reduction relation induced by a rewrite system may order the rules using criteria like computational efficiency or ease of programming. Also, each equivalence class of terms has a unique irreducible term to represent it; this irreducible term is considered to be the normal form of the equivalence class (or each term in the equivalence class). Formally, a rewrite system R is *confluent* iff for all terms s and t such that $s \Leftrightarrow_R^* t$, there is a term v for which $s \Rightarrow_R^* v$ and $t \Rightarrow_R^* v$.

A weaker version of confluence is local confluence: a rewrite system R is *locally confluent* iff for all terms s, t, and u such that $s \Rightarrow_R t$ and $s \Rightarrow_R u$, there is a term v for which $t \Rightarrow_R^* v$ and $u \Rightarrow_R^* v$. Since $s \Rightarrow_R t$ and $s \Rightarrow_R u$ implies $t \Leftrightarrow_R^* u$, it follows that a confluent rewrite system is also locally confluent.

Termination: Every sequence of rule applications should terminate. Thus, each term can be reduced by using a terminating rewrite system to a normal form in a finite number of rewriting steps. Consequently, any process based on a terminating rewrite system always terminates. Proving confluence for terminating systems, which is usually done using an approach first presented in [KB70], is easier than for non-terminating systems. Formally, a rewrite system R is *terminating* if there is no infinite chain $t_1 \Rightarrow_R t_2 \Rightarrow_R \ldots$ of terms.

A rewrite system R is *convergent* if it is both confluent and terminating. For a convergent system, all rewrite sequences terminate after a finite number of steps producing a unique normal form. Thus, the reduction relation $\Rightarrow_R^!$ induced by a convergent rewrite system R is a function, usually denoted by RF, on the set of terms. Thus, for any terms s and t, $RF(s) = t$ iff $s \Rightarrow_R^! t$.

Modularity: Parts of a theory should be independently rewritable before rewriting the entire theory. Since theories that represent knowledge bases are usually built incrementally, a modular rewrite system allows reuse of the reduced form of the original theory for reasoning with the new knowledge base. Formally, a rewrite system R is *modular (with respect to $\odot$)* iff for any bags B, B_1, and B_2 of formulas, if $\odot(B_1) \Leftrightarrow_R^* \odot(B_2)$ then $\odot(B, B_1) \Leftrightarrow_R^* \odot(B, B_2)$.

Monotonicity: Rewriting should not shrink the set of facts directly inferable from a theory. A rewrite system that is not monotonic is not suitable for reasoning since fewer facts may be inferable after rewriting. Formally, a rewrite system R is *monotonic (with respect to facts)* iff for any logical terms s and t, if $s \Rightarrow_R^* t$ then $\text{facts}(s)^{\doteq} \subseteq \text{facts}(t)^{\doteq}$.

Preservation: Since rewriting will be used for logical reasoning, it should not change the logical content of a theory or formula. Formally, a rewrite system R is *(content) preserving* iff for any logical terms s and t, if $s \Leftrightarrow_R^* t$ then $s \equiv t$. Thus, if theory Γ rewrites to theory Δ in possibly several steps then Γ and Δ are logically equivalent.

Tractability: It should be computationally tractable to rewrite any theory to an irreducible form, since this ensures that reasoning based on rewriting is tractable. Formally, a rewrite system R is *tractable* iff there is a PTIME algorithm which given any logical term s as input, outputs a logical term t such that $s \Rightarrow_R^! t$.

The next lemma shows the effect of these properties on the reduction function induced by a convergent rewrite system for theories:

Lemma 2.1 *For the reduction function, RF, induced by any convergent rewrite system R for PCE, and any theories Γ and Δ:*

1. *if R is content preserving, then $\Gamma \equiv RF(\Gamma)$;*
2. *if R is monotonic with respect to facts, then* $\text{facts}(\Gamma) \subseteq \text{facts}(RF(\Gamma))$*;*
3. *if R is modular with respect to $\odot$ then $RF(\Gamma \cup \Delta) = RF(RF(\Gamma) \cup \Delta)$;*
4. *if R is tractable, then there is a PTIME algorithm which returns $RF(\Gamma)$ for any input theory Γ.*

Proof: RF is defined since R is convergent. Since $\Gamma \Rightarrow_R^! RF(\Gamma)$, all the claims follows directly from the definitions. ■

We will usually drop the suffixes "with respect to $\odot$" and "with respect to facts" when there is no possibility of confusion.

We will propose some rewrite systems based on various desiderata, but will then modify them if they do not satisfy the properties listed above. Typical modifications include adding a new rule, removing a rule, or modifying a rule. As a trivial example, any rewrite system can be extended to a confluent system by adding to it the inverse of each rewrite rule in it. However, this will produce a non-terminating system as there would be no irreducible terms and thus would not be of much interest to us. In general, we will seek rewrite systems that satisfy all the above properties.

2.4.3 Proof Techniques

The properties presented in the previous section are global to the entire rewrite system. In this section, we present some sufficient local conditions, which are defined in terms of individual rules or pairs of rules, for proving these properties of a rewrite system. This subsection may be skimmed in a first reading; it becomes relevant mostly when proofs of various properties are carried out.

The simplest properties to prove are content preservation, monotonicity, and modularity. The next three lemmas show that these properties can be proved by verifying simple properties for individual rewrite rules. Termination is usually proved by showing that there is a well-founded ordering $>$ on terms such that $l > r$ for each rewrite rule $l \Rightarrow r$. For proving confluence, we will show that each overlap term with respect to each pair of rewrite rules reduces to the same term after either rule application. Tractability is proved by presenting a tractable algorithm that outputs the irreducible form of the input term.

Lemma 2.2 *A rewrite system R is content preserving if $l \equiv r$ for each rewrite rule (instance) $l \Rightarrow r$ in R.*

Proof: For any interpretation, the truth value of a logical term is defined using the truth values of its proper logical subterms; there is no other dependency on the subterms. Thus, replacing a subterm by a logically equivalent term does not change the truth value of the term, i.e., if $s \Rightarrow_R t$ then $s \equiv t$. Since $\equiv$ is an equivalence relation, the claim follows. ■

Lemma 2.3 *A rewrite system R is monotonic with respect to facts if* $\text{facts}(l)^{\dot{=}} \subseteq \text{facts}(r)^{\dot{=}}$ *for each rewrite rule (instance) $l \Rightarrow r$ in R.*

Proof: For any term s, facts(s) is monotonic in each of facts(l), where l is any subterm of s. Thus, if t is obtained by replacing a subterm l in s by r, i.e., $s \Rightarrow_R t$, and if $\text{facts}(l)^{\dot{=}} \subseteq \text{facts}(r)^{\dot{=}}$, then $\text{facts}(s)^{\dot{=}} \subseteq \text{facts}(t)^{\dot{=}}$. Since $\subseteq$ is transitive, the claim follows. ■

Note that $\text{facts}(l) \subseteq \text{facts}(r)$ is a sufficient condition for ensuring that $\text{facts}(l)^{\dot{=}} \subseteq \text{facts}(r)^{\dot{=}}$.

Lemma 2.4 *A rewrite system R is modular with respect to $\odot$ if $\odot(B, B_1) \Leftrightarrow_R^* \odot(B, B_2)$ for each rewrite rule (instance) $\odot(B_1) \Rightarrow \odot(B_2)$ in R and each bag B of formulas.*

Proof: Modularity requires that for any bags B, B_1, and B_2 of formulas, if $\odot(B_1) \Leftrightarrow_R^* \odot(B_2)$ then $\odot(B, B_1) \Leftrightarrow_R^* \odot(B, B_2)$. We will prove this by showing that each step in $\odot(B_1) \Leftrightarrow_R^* \odot(B_2)$ has a corresponding step in $\odot(B, B_1) \Leftrightarrow_R^* \odot(B, B_2)$

Suppose $\odot(B_1) \Rightarrow_R \odot(B_2)$ using some instance of rule R_i. If R_i is a formula rule then $\odot(B, B_1) \Rightarrow_R \odot(B, B_2)$ using the same rule instance of R_i, because $\odot(B, B_1)$ has as formula subterms all the formula subterms of $\odot(B_1)$. Otherwise R_i is a theory rule schema, and since the constructor $\odot()$ cannot appear nested, $\odot(B_1) \Rightarrow \odot(B_2)$ is actually an instance of R_i; the result then follows from the hypothesis of the lemma. ■

Proving Termination

Termination of a rewrite system is typically proved by defining a well-ordering on terms and then showing that each rewrite step produces a "smaller" term with respect to this ordering. Since this can only happen a finite number of times, every sequence of rewriting terminates. However, the ordering should satisfy certain properties, which are analogous to the notion of termination orderings [DJ90]:

Definition 2.17 An ordering $>$ on terms is <u>*closed under replacement*</u> iff for any terms s, t, p, q: if $p > q$ and s is a subterm of t at position π then $t_\pi[p \leftarrow s] > t_\pi[q \leftarrow s]$. An ordering $>$ on terms is <u>*compatible with equality*</u> iff for any terms s, t, s', t': if $s = s'$ and $t = t'$, then $s > t$ iff $s' > t'$. ■

Intuitively, if an ordering $>$ is closed under replacement and $s \Rightarrow_R t$ using a rewrite rule instance $l \Rightarrow r$ such that $l > r$ then $s > t$. Also, if $>$ is compatible with equality then a term can be replaced by a syntactically equivalent term without effecting its order with respect to other terms. The next lemma shows how these properties can be used for proving termination.

Lemma 2.5 (based on [DJ90]) *A rewrite system R is terminating if there is a well-ordering $>$ on terms such that $>$ is closed under replacement, $>$ is compatible with equality, and $l > r$ for each rewrite rule (instance) $l \Rightarrow r$ in R.*

Proof: Suppose $s \Rightarrow_R t$, i.e., there is a rewrite rule $l \Rightarrow r$ in R such that l is a subterm of s and $t = s_\pi[r \leftarrow l]$. Since $>$ is closed under replacement and $l > r$, $s_\pi[l \leftarrow l] > s_\pi[r \leftarrow l]$, i.e., $s > s_\pi[r \leftarrow l]$. Thus, $s > t$, since $>$ is compatible with equality. Since $>$ is a well-ordering, there is no infinite chain $t_1 \Rightarrow_R t_2 \Rightarrow_R \ldots$ of terms. Thus, R is terminating. ■

We now present a specific ordering that will be used for proving the termination of FP and other rewrite systems presented later. It extends the total well-ordering $\succ$ on the set $\mathcal{C}$ of constants to a well-ordering on all the terms.

As is usual, the ordering $\succ$ among constants can be extended to the <u>*multiset ordering*</u> $\succ_{mul}$ among bags of constants: $B \succ_{mul} B'$ iff (1) $B \neq B'$, and (2) whenever $B'(b) > B(b)$ for some constant b then there is a constant $a \succ b$ such that $B(a) > B'(a)$. Recall that $B(b)$ denotes the number of occurrences of the element

b in the bag B. For example, if $a \succ b$, then $[\![a, b, a]\!] \succ_{mul} [\![b, a, b, b]\!]$. Since $\succ$ is a well-ordering, $\succ_{mul}$ is also a well-ordering [DJ90].

We now use this multiset ordering and the number of occurrences of literals and connectives to extend the well-ordering $\succ$ to the set of terms. For this, we first define four functions on terms: w_1, which counts the number of occurrences of literals, w_2, which counts the number of occurrences of connectives, w_3, which is a weighted sum of literals, and w_4, which collects all constants in a bag.

Definition 2.18 The functions w_1, w_2, w_3, and w_4 on the set of all terms are defined recursively as follows:

1. for any constant a: $w_1(a) = w_2(a) = w_3(a) = 0$;
2. for any literal α, $w_1(\alpha) = 1$, $w_2(\alpha) = 0$, and $w_3(\alpha) = 1$;
3. for any connective c and any bag B of formulas:

$$\begin{aligned} w_1(c(B)) &= \sum_{\psi \in B} w_1(\psi) \\ w_2(c(B)) &= 1 + \sum_{\psi \in B} w_2(\psi) \\ w_3(c(B)) &= 2 \times \sum_{\psi \in B} w_3(\psi) \end{aligned}$$

4. for any term t, $w_4(t)$ is the bag of all the constants (including repetitions) that are subterms of t.

For any terms s and t: $s \succ t$ iff

1. either $w_1(s) > w_1(t)$;
2. or $w_1(s) = w_1(t)$ and $w_2(s) > w_2(t)$;
3. or $w_1(s) = w_1(t)$, $w_2(s) = w_2(t)$, and $w_3(s) > w_3(t)$;
4. or $w_1(s) = w_1(t)$, $w_2(s) = w_2(t)$, $w_3(s) = w_3(t)$, and $w_4(s) \succ_{mul} w_4(t)$.

∎

For example, if term $t = \vee(\wedge(P, a \doteq b), \neg R(a, b, c))$ then $w_1(t) = 3$, $w_2(t) = 2$, $w_3(t) = 2(2(1+1)+1) = 10$, and $w_4(t) = [\![a, a, b, b, c]\!]$. The next proposition shows that $\succ$ is suitable for proving termination of a rewrite system by using Lemma 2.5.

Proposition 2.6 *The ordering $\succ$ on terms is a well-ordering, is closed under replacement, and is compatible with equality.*

Proof: Since $\succ$ among terms is a lexicographic combination of four well-orderings, it follows that $\succ$ among terms is also a well-ordering [DJ90].

For any fixed subterm s of term t, for each $j = 1 \ldots 3$, if $w_j(q) > w_j(p)$ then $w_j(t_\pi[q \leftarrow s]) > w_j(t_\pi[p \leftarrow s])$; moreover, if $w_4(q) \succ_{mul} w_4(p)$ then $w_4(t_\pi[q \leftarrow s]) \succ_{mul} w_4(t_\pi[p \leftarrow s])$. Thus, $\succ$ is closed under replacement.

If two terms are equal, i.e., $s = t$, then one must be obtainable from the other by reordering its arguments. Since the ordering $\succ$ among terms does not depend on the ordering of arguments, $>$ is compatible with respect to equality. ∎

Proving Confluence

A terminating rewrite system can be proved confluent using the approach first presented in [KB70]. The basic idea is to prove local confluence, i.e., if a term s can be rewritten to two distinct terms t_1 and t_2 in one rewriting step then there is a common term t such that both t_1 and t_2 rewrite to t, possibly using many rewriting steps. Local confluence and termination guarantees confluence. Further, it is not necessary to show local confluence for all such triples $\langle s, t_1, t_2 \rangle$; it is sufficient to consider only the cases where t_1 and t_2 are obtained by using rules which are applied at "overlapping" positions (as explained below).

For example, consider the rewrite system R containing the following rule schemas:

$$
\begin{array}{lrcl}
R_1: & x \doteq x & \Rightarrow & \mathbf{t} \\
R_2: & \vee(\vee(\alpha, B)) & \Rightarrow & \vee(\alpha, B) \\
R_3: & \wedge(\alpha, B) & \Rightarrow & \wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])
\end{array}
$$

where the meta-variables x, α, and B can be instantiated by constants, literals, and bags of formulas, respectively.

Consider the term $s_1 = \vee(a \doteq a, b \doteq b)$, where a and b are constants. Term s_1 rewrites to the term $\vee(\mathbf{t}, b \doteq b)$ using the instance $a \doteq a \Rightarrow \mathbf{t}$ of rule R_1, and to the term $\vee(a \doteq a, \mathbf{t})$ using another instance $b \doteq b \Rightarrow \mathbf{t}$ of the same rule. Clearly, both of these rewrite to the term $\vee(\mathbf{t}, \mathbf{t})$. Note that the left-hand sides of these two rule instances match distinct, non-overlapping subterms of s_1. As proved later in Lemma 2.7, the two rule applications in such cases can be performed in either order, and produce the same final result.

Now consider a different term, $s_2 = \vee(\vee(P, a \doteq a))$, where P is an atom and a is a constant. Term s_2 rewrites to the term $t_1 = \vee(\vee(P, \mathbf{t}))$ using the instance $a \doteq a \Rightarrow \mathbf{t}$ of rule R_1 at position 12. Term s_2 also rewrites to the term $t_2 = \vee(P, a \doteq a)$ using the R_2-instance $\vee(\vee(P, a \doteq a)) \Rightarrow \vee(P, a \doteq a)$ at position Λ. (In this particular case, t_1 and t_2 both can be rewritten to $t = \vee(P, \mathbf{t})$.) Note that subterms of s_2 that match the left-hand sides of the two rule instances are not distinct: the position Λ is a prefix of 12. In the standard terminology of rewrite systems, the term s_2 is called an *overlap* (at position 1.2) between the R_1-instance and the R_2-instance, which are called the *outer* and the *inner* rules, respectively. The pair t_1 and t_2 of terms is called a *critical pair*, and the term t is a *common term* to which the critical pair rewrites. In some cases, rewriting the critical pair to a common term requires rules in R which are not instances of the two rules (R_1 and R_2, in this case) used to obtain the critical pair — these rules are called the *extra rules*.

By focussing only on overlaps, we now define a notion of confluence for pairs of rules in a rewrite system:

Definition 2.19 (see Figure 2.2) In a rewrite systems R, a rule instance $R_1 = (l_1 \Rightarrow r_1)$ is confluent with rule instance $R_2 = (l_2 \Rightarrow r_2)$, if whenever l_2 is a subterm of l_1 at some position π, then there is some r such that $r_1 \Rightarrow_R^* r$ and $l_{1\,\pi}[r_2 \leftarrow l_2] \Rightarrow_R^* r$.[2] ∎

The significance of this definition becomes evident in the following result:

Lemma 2.7 (based on [KB70]) *A terminating rewrite system R is confluent if each pair of rule instances in R is confluent.*

Proof: Any terminating rewrite system R is confluent iff it is locally confluent [New42]. Hence, it suffices to prove local confluence.

Suppose $t \Rightarrow_R t_1$ using rule $R_1 = (l_1 \Rightarrow r_1)$ and $t \Rightarrow_R t_2$ using rule $R_2 = (l_2 \Rightarrow r_2)$. Thus, there are positions π_1 and π_2 in t such that $t_1 = t_{\pi_1}[r_1 \leftarrow l_1]$, and $t_2 = t_{\pi_2}[r_2 \leftarrow l_2]$. We obtain the desired t' in all possible cases:

distinct: if neither π_1 nor π_2 is a prefix of the other (i.e., the terms matching l_1 and l_2 do not overlap in t, which is a tree-like structure), then each of t_1 and t_2 rewrite to $t' = (t_{\pi_1}[r_1 \leftarrow l_1])_{\pi_2}[r_2 \leftarrow l_2]$ using rules R_2 and R_1, respectively.

[2] Note that the relation "confluent with" is asymmetric.

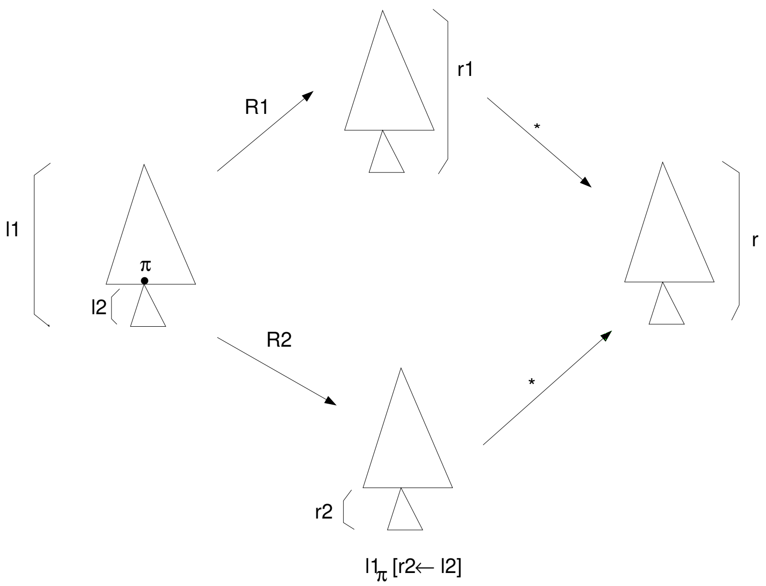

Figure 2.2: Overlap, critical pair, common term, and confluence

overlapping: If the subterms matching l_1 and l_2 overlap, then without loss of generality, assume that l_2 is a subterm of l_1 (recall that these are rule instances, with no variables of any kind). Since nothing outside l_1 in t is changed in obtaining either t_1 or t_2, we can also assume, without loss of generality, that $t = l_1$. Thus, $t_1 = r_1$, and the result follows by the hypothesized confluence of the rules.

■

Our general technique to prove the confluence of a terminating rewrite system R is to take every pair of *rule schemas* R_1 and R_2, identify all the ways in which an instance $l_2 \Rightarrow r_2$ of R_2 has the property that l_2 is a subterm of l_1, for some instance $l_1 \Rightarrow r_1$ of R_1, and show that $l_1 \Rightarrow r_1$ is confluent with $l_2 \Rightarrow r_2$, by showing how the critical pairs rewrite to a common term. Since confluence among rules is not symmetric, we need to check both that instances of R_1 are confluent with that of R_2, and that instances of R_2 are confluent with that of R_1. We also have to verify that instances of R_1 are confluent with those of R_1 itself. (The reader is cautioned that there are two differences from the approach in [KB70]: we cannot ignore all "variable overlaps", and we will use various techniques to reduce the cases of non-variable overlap to be considered.)

We have been and will continue to treat rule schemas by taking prototypical instances of them, and seeing how they can overlap (remembering the conditions imposed on the possible instantiations). There is one circumstance however where it will be useful to again distinguish rule schemas from instances.

Definition 2.20 (see Figure 2.3) A *variable overlap* is an overlap in l_1 at position π between an R_1-instance $l_1 \Rightarrow r_1$ and an R_2-instance $l_2 \Rightarrow r_2$ such that there is a meta-variable at some prefix of position π in the left-hand side of rule schema R_1. ■

Thus, a variable overlap means that l_2 is a subterm of the term used to instantiate the meta-variable in the schema R_1 for obtaining the instance $l_1 \Rightarrow r_1$. In the example given earlier, s_2 is indeed a variable overlap. Typically, in rewrite systems [KB70] variable overlaps are ignored while proving confluence, since the two rules can be interchanged to obtain the same common term (as illustrated by s_2). However, we cannot ignore variable overlaps for two reasons: there are restrictions on how the meta-variables are instantiated in

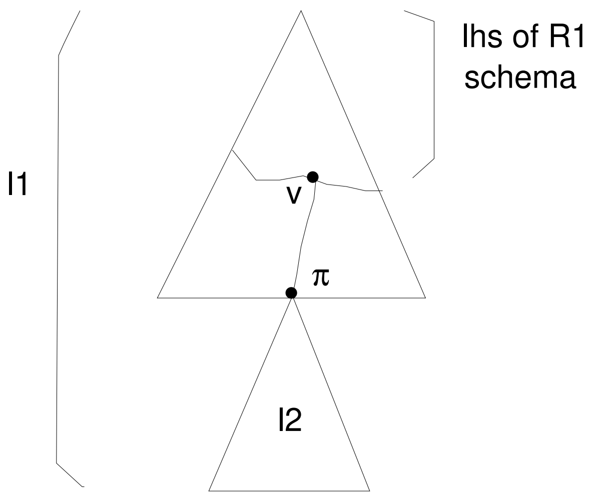

Figure 2.3: Variable overlap: v is a meta-variable in R_1's schema

obtaining rule instances, and the terms instantiated for the meta-variables in the left-hand side of the rule may be modified in the right-hand sides. For example

1. Consider the term $t = \vee(\vee(a \doteq a, \wedge(P)))$ and the variable overlap at position 1.1 between an R_2-instance $\vee(\vee(a \doteq a, \wedge(P))) \Rightarrow \vee(a \doteq a, \wedge(P))$ and an R_1-instance $a \doteq a \Rightarrow \mathbf{t}$. Although both the rule instances apply to t, if the R_1 instance is first applied then the R_2 instance does not apply (since R_2 requires that α be a literal and $\mathbf{t}$ is not a literal). Such variable overlaps cannot be ignored.

2. Consider the term $t = \wedge(P, \vee(\vee(P, \wedge(q))))$ and the variable overlap at position 2 between an R_3-instance $\wedge(P, \vee(\vee(P, \wedge(q)))) \Rightarrow \wedge(P, \vee(\vee(\mathbf{t}, \wedge(q))))$ and an R_2-instance $\vee(\vee(P, \wedge(q))) \Rightarrow \vee(P, \wedge(q))$. Although both the rule instances apply to t, if the R_3-instance is applied first then the R_2-instance is no longer applicable. Such variable overlaps also cannot be ignored.

Fortunately, as illustrated by s_2, there are still a large number of variable overlaps that can be ignored in our case as well. The next definition and proposition are motivated to identify the variable overlaps that can be ignored.

Definition 2.21 A variable overlap at position π between an R_1-instance $l_1 \Rightarrow r_1$ and an R_2-instance $l_2 \Rightarrow r_2$ is *non-conflicting* if:

1. the meta-variable at a prefix of position π in the left-hand side of rule schema R_1 is not modified and appears at most once in its right-hand side, and

2. ${l_1}_\pi[r_2 \leftarrow l_2]$ is the left-hand side of an instance of rule R_1.

■

The next propositions shows that we can ignore non-conflicting variable overlaps in proving that a pair of rules is confluent.

Proposition 2.8 *Any critical pair for a non-conflicting variable overlap is rewritable to a common term.*

Proof: Suppose the variable overlap is at position π between an R_1-instance $l_1 \Rightarrow r_1$ and an R_2-instance $l_2 \Rightarrow r_2$. The critical pair is r_1 and ${l_1}_\pi[r_2 \leftarrow l_2]$. Suppose V is the meta-variable at a prefix of position π in the left-hand side of rule schema R_1. We know that V is not modified in the right-hand side of schema R_1. Now there are two cases:

1. V appears on the right-hand side of schema R_1, say at position π': Since l_2 must be a subterm of $r_1|_{\pi'}$, rule R_2 rewrites r_1 to ${r_1}_\pi[r_2 \leftarrow l_2]$, which is the desired common term.
2. V does not appear in the right-hand side of schema R_1: the desired common term is r_1.

■

Another way to reduce our work is to use a "symmetry" grouping of rules based on the notions of "duality" and "cduality", introduced next:

Definition 2.22 The mapping <u>*dual*</u>, denoted by the superscript ^{d}, on constants and functors, is defined to be identity everywhere except for $\wedge^d = \vee, \vee^d = \wedge$ and $\doteq^d = \not\doteq$, $\not\doteq^d = \doteq$. This mapping extends naturally to a morphism over terms (hence to formulas, and theories), as well as rules, with $(l \Rightarrow r)^d$ being equal to $l^d \Rightarrow r^d$.

The mapping <u>*cdual*</u>, denoted by the superscript ^{c}, on terms is defined to be identity everywhere except for $\odot(B)^c$ being equal to $\wedge(B)$, where B is any bag of formulas. This mapping again extends to a morphism over rules, with $(l \Rightarrow r)^c$ being equal to $l^c \Rightarrow r^c$. ■

Observe that for every term t, $(t^d)^d = t$, and that $\mathbf{t}^d = \mathbf{f}, \mathbf{f}^d = \mathbf{t}$. However, $(t^c)^c = t^c$.

Definition 2.23 A rewrite system R is <u>*closed with respect to duals*</u> iff the dual of each rule in R is in R. A rewrite system R is <u>*closed with respect to cduals*</u> iff the cdual of each rule in R is in R. ■

The notions of dual and cdual will simplify the proofs for confluence.

Proposition 2.9 *For any rewrite system R closed with respect to duals, if rule R_1 is confluent with R_2, then R_1^d is confluent with R_2^d.*

Proof: The proof is based on the fact that every sequence of rewrites has a corresponding "dual", which follows from the fact that if $s \Rightarrow_R u$ using some rule $l \Rightarrow r$ (i.e., $u = s_\pi[r \leftarrow l]$), then $s^d \Rightarrow_R u^d$ using the dual of the rule, because l^d is a subterm of s^d at position π iff l is a subterm of s at position π, and because $(s_\pi[r \leftarrow l])^d = {s^d}_\pi[r^d \leftarrow l^d]$.

Hence, suppose that for some overlap t^d of R_1^d with R_2^d, the resulting critical pair is t_1^d and t_2^d. Then by the definition of duality for terms, t is an overlap for R_1 with R_2, having critical pair t_1 and t_2. By the confluence of R_1 with R_2, these can be both rewritten to some common term t_3. Since all duals of rules are rules in R, t_1^d and t_2^d can also be rewritten to t_3^d, which is then the common term we are seeking. ■

Proposition 2.10 *For any rewrite system, if rule R_1 is confluent with R_2 and if each extra rule used in rewriting the critical pairs to common terms has a cdual in R, then also: R_1 is confluent with R_2^c, R_1^c is confluent with R_2, and R_1^c is confluent with R_2^c.*

Proof: The proof is similar to that of Proposition 2.9. Given that rule R_1 is confluent with R_2, we will prove that R_1^c is confluent with R_2^c; the other claims can be similarly proved.

Suppose that for some overlap t^c of R_1^c with R_2^c, the resulting critical pair is t_1^c and t_2^c. By the definition of cduality for terms, t is an overlap for R_1 with R_2, having critical pair t_1 and t_2. By the confluence of R_1 with R_2, these can be both rewritten to some common term t_3. Since all cduals of extra rules are rules in R, t_1^c and t_2^c can also be rewritten to t_3^c, which is then the common term we are seeking. ■

It follows that for any rewrite system R closed with respect to cduals, if rule R_1 is confluent with R_2 then R_1' is confluent with R_2', where each R_i' is either R_i or R_i^c.

It is important to note that the above results require rule *instances* to have duals and cduals, not just rule schemas. Therefore in specifying duals and cduals for rule schemas we must make sure that conditions for instantiating meta-variables cannot be violated by the process of dualization and cdualization.

In order to prove confluence, we will show that each pair of rules is confluent (Lemma 2.7). The number of pairs to be considered will be reduced using Propositions 2.9, 2.10, and 2.8.

2.5 BCP : Boolean Constraint Propagation

We review variants of boolean constraint propagation (BCP) [McA80, McA90] that have been proposed in the literature. We show that some of them can be specified using simple rewrite systems. Recall that BCP is an incomplete method for simplifying finite clausal theories in propositional logic and for inferring facts entailed by a given theory. As is usual in descriptions of BCP, we restrict our attention to the finite PC, a syntactic variant of PCE without equality. Thus, there is no equality predicate in formulas and theories, and all the atoms are propositions.

We will first define a measure of complexity on algorithms for inferring facts from a theory. We will then discuss Horn pebbling, which is BCP restricted to Horn clauses, and clausal BCP, which allows arbitrary clauses. We will also review three extensions of clausal BCP to non-clausal theories.

Given any algorithm A for inferring facts from a theory, we define the following decision problem for inferring facts using A:

Definition 2.24 The *Fact-inference problem for A* is defined to be the following decision problem:

Input:	any theory Γ and any fact α;
Output:	"yes" iff Algorithm A infers fact α from theory Γ.

■

Note that there is a straight-forward algorithm for solving the fact-inference problem for A, namely, scan the list of facts inferred by A and return "yes" iff α is in the list. However, an algorithm for solving the fact-inference problem for A does not have to generate all the facts inferred by A from the theory. It can also use the fact α, which is part of the input, to work in a goal-directed fashion. Thus, the time complexity of the fact-inference problem for A could be lower than the time complexity of the algorithm A itself.

2.5.1 Horn Pebbling

Dowling and Gallier [DG84] developed a linear-time algorithm for determining satisfiability of Horn theories that also obtains all the positive literals that are logically entailed by the input theory. We will first present this "Horn Pebbling" algorithm and then specify it using an inference system and a rewrite system.

Horn Pebbling works by pebbling a labeled directed graph whose nodes represent atoms and truth constants, and whose arcs encode Horn clauses (named by integers). The result of the pebbling is that node **f** is pebbled iff the given theory is unsatisfiable; otherwise, exactly those nodes are pebbled which represent atoms that are logically entailed by the theory.

For each atom in the theory, there is a node in the *pebbling graph* labeled by that atom; there is also a node labeled by **t** and a node labeled by **f**. Given any ordering of clauses in the theory, the edges of the graph are as follows:

1. if the ith clause is $\vee(P)$, for some positive literal P, then there is an edge labeled by i from **t** to P.
2. if the ith clause is $\vee(\neg P_1, \ldots, \neg P_n)$, then for each node labeled P_j, where $1 \leq j \leq n$, there is an edge from P_j to **f** labeled by i.

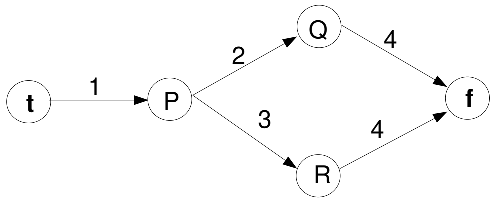

Figure 2.4: A pebbling graph

3. if the ith clause is $\vee(P, \neg P_1, \ldots, \neg P_n)$, then for each node labeled P_j, where $1 \le j \le n$, there is an edge from P_j to P labeled by i.

For any node u in the graph and any i, any node w such that there is an edge labeled i from w to u is called an *i-antecedent* of u. The *pebbling game*, used for pebbling the nodes of the graph, proceeds as follows:

1. the node labeled by $\mathbf{t}$ can be pebbled any time;
2. if the node labeled by $\mathbf{f}$ is pebbled, then any other node can be pebbled;
3. if u is a node with an i-antecedent node such that *all* i-antecedent nodes of u are pebbled, then u can be pebbled.

A *pebbling sequence* is a sequence of labels of nodes which are pebbled in some run of the pebbling game. Each fact in a pebbling sequence is said to be *inferred* from the theory using Horn pebbling.

For example, consider the Horn theory Γ consisting of the following clauses:

1. (P)
2. $(\neg P \vee Q)$
3. $(\neg P \vee R)$
4. $(\neg Q \vee \neg R)$

The pebbling graph for Γ is given in Figure 2.4. In the pebbling game, after pebbling the node labeled by $\mathbf{t}$ and then the node labeled by P, the nodes labeled Q and R can be pebbled in any order, and then the node labeled $\mathbf{f}$ is pebbled. Thus, there are two pebbling sequences — $\mathbf{t}, P, Q, R, \mathbf{f}$ and $\mathbf{t}, P, R, Q, \mathbf{f}$ — each ending with a pebble on the node labeled by $\mathbf{f}$. Intuitively, Horn Pebbling can determine that Γ is unsatisfiable.

Consider the Horn theory Δ consisting of the first three clauses of Γ. The pebbling graph for Δ is identical to that of Γ, without the edges labeled 4. Any run of the pebbling game pebbles exactly the nodes labeled by $\mathbf{t}, P$, and Q. Thus, Horn pebbling infers the facts $\mathbf{t}, P$, and Q from the theory Δ.

It turns out that each fact inferred from a theory using Horn pebbling is logically entailed by the theory. However, Horn Pebbling cannot be used to obtain any negative literal that is logically entailed by a Horn theory; for example, the literal $\neg P$ that is entailed by the theory $[\![(\neg P \vee Q), (\neg P \vee R), (\neg Q \vee \neg R)]\!]$ cannot be inferred using Horn Pebbling, since the pebbling game stops after pebbling the node labeled by $\mathbf{t}$.

An efficient algorithm for implementing the pebbling game is presented in [DG84]. If all the atoms that appear in the theory are known a priori (say, in a list provided as part of the input), then the algorithm terminates in time $O(n)$, where n is the total number of occurrences of literals in the theory.

The following inference rule, adapted from [de 89], provides a deductive system for the *facts* that are inferred from a theory using Horn Pebbling:

$$\frac{\vee(p);\ \vee(\neg p, \alpha_1, \ldots, \alpha_n)}{\vee(\alpha_1, \ldots, \alpha_n)} \quad \cdots\cdots\cdots \quad \mathbf{DHP}$$

$$\begin{aligned} \odot(\mathbf{f}, B) &\Rightarrow \odot(\mathbf{f}) \\ \odot(\vee(p), \vee(\sim p, B_1), B_2) &\Rightarrow \odot(\vee(p), \vee(B_1), B_2) \end{aligned}$$

where p is an atom, B_i are bags of formulas, and B is a non-empty bag of formulas.

Figure 2.5: A rewrite system, HP, for Horn Pebbling

$$\begin{aligned} \odot(\mathbf{f}, B) &\Rightarrow \odot(\mathbf{f}) \\ \odot(\vee(\alpha), \vee(\sim \alpha, B_1), B_2) &\Rightarrow \odot(\vee(\alpha), \vee(B_1), B_2) \end{aligned}$$

where α is a literal and B's are bags of formulas; B must be non-empty.

Figure 2.6: A rewrite system, CBCP, for Clausal BCP

This rule is merely modus ponens with one antecedent restricted to an atom and the other to a clause (recall that $\vee(p)$ is a clausal representation of the atom p). An atom p in a theory Γ is pebbled iff either $\vee(p)$ or $\mathbf{f}$ is inferable from Γ using the above inference rule. As observed by [de 89], the second antecedent clause, which is a superclause of the consequence clause in the inference rule, may be removed from the theory after this rule is applied, since its further use can be replaced by the consequence clause.

The rewrite system, HP, of Figure 2.5 provides an alternative characterization of Horn Pebbling:

Proposition 2.11 *For any Horn theory Γ and any clausal fact ψ, ψ is inferable from Γ using DHP iff there is a clause ψ' that is a subclause of ψ and a bag B of clauses such that $\Gamma \Rightarrow^*_{HP} \odot(\psi', B)$.*

Proof: By simple inductions on the length of proof using DHP and the length of the rewriting sequence. Only the second rewrite rule is required for this proof. We need "subclause" since the antecedent clause of the inference rule is explicitly discarded in the rewrite rule. ∎

The first rewrite rule ensures that HP is convergent. Note that once $\mathbf{f}$ is pebbled, all facts are derivable. It then follows that the set of facts inferable from a theory Γ using Horn Pebbling is exactly facts(Γ'), where $\Gamma \Rightarrow^!_{HP} \Gamma'$.

2.5.2 Clausal BCP

Clausal BCP [McA80] generalizes Horn Pebbling to any clausal theory. Its worst-case time complexity is the same as Horn Pebbling, i.e., $O(n)$ when all the atoms are known a priori. However, it is strictly more powerful than Horn Pebbling, i.e., it obtains all the facts that are obtained by Horn Pebbling, and occasionally more (e.g., negative literals). After defining Clausal BCP, we characterize it using an inference system and a rewrite system.

Given any clausal theory Γ, Clausal BCP monotonically expands it by adding facts as follows: in each step, if any *single* clause in Γ and *all the facts* in Γ taken together *logically entail* any other fact, then the new fact is added to the theory Γ. This step is repeated until no new fact can be so obtained. The facts in the resulting theory are said to be inferred from Γ using Clausal BCP. Each step of Clausal BCP can be efficiently carried out, since the only ways in which facts are entailed from a clause and other facts are:

1. clause $\vee(\alpha, \sim\alpha_1, \ldots, \sim\alpha_n)$ and all the facts $\vee(\alpha_i)$ (where $1 \le i \le n$) logically entail the fact $\vee(\alpha)$, and

2. clause $\vee(\sim\alpha_1, \ldots, \sim\alpha_n)$ and all the facts $\vee(\alpha_i)$ (where $1 \le i \le n$) logically entail any fact.

Consider the clausal theory $\Delta = [\![(\neg P), (P \vee \neg Q), (P \vee \neg R), (Q \vee R)]\!]$. Since clause $(Q \vee R)$ contains more than one positive literal, Δ is not a Horn theory. A sequence of facts added to Δ using Clausal BCP is $(\neg P), (\neg Q), (\neg R), \mathbf{f}$. Thus, Clausal BCP can determine that this theory is unsatisfiable. Note that Horn Pebbling would not pebble any node, and thus, cannot determine this. However, Clausal BCP is incomplete even for positive literals; for example, the literal R that is entailed by the theory $[\![(P \vee Q), (\neg P \vee R), (\neg Q \vee R)]\!]$ cannot be inferred using Clausal BCP.

Adapting from [de 89] again, the following inference rule provides a deductive system for facts inferred by Clausal BCP:

$$\frac{\vee(\alpha);\ \vee(\sim\alpha, \alpha_1, \ldots, \alpha_n)}{\vee(\alpha_1, \ldots, \alpha_n)}$$

This generalizes the inference rule for Horn Pebbling, since α is now allowed to be a negative literal also. As in the case of Horn Pebbling, the second antecedent clause in the above inference rule can be removed from the theory after this rule is applied, if all we care about are the facts deduced.

The rewrite system, CBCP, given in Figure 2.6 provides an alternative characterization of Clausal BCP: the set of facts inferable from a theory Γ using Clausal BCP is exactly facts(Γ'), where $\Gamma \Rightarrow^{!}_{CBCP} \Gamma'$. As with Horn Pebbling, CBCP is equivalent to the inference system in terms of inferable facts.

For example, using the rewrite system CBCP, the theory $\Delta = [\![(\neg P), (P \vee \neg Q), (P \vee \neg R), (Q \vee R)]\!]$, given above, may be reduced in the following sequence:

$$\begin{array}{ll}
\Rightarrow_{CBCP} & [\![(\neg P), (\neg Q), (P \vee \neg R), (Q \vee R)]\!] \quad \text{(2nd rule)} \\
\Rightarrow_{CBCP} & [\![(\neg P), (\neg Q), (\neg R), (Q \vee R)]\!] \quad \text{(2nd rule)} \\
\Rightarrow_{CBCP} & [\![(\neg P), (\neg Q), (\neg R), (R)]\!] \quad \text{(2nd rule)} \\
\Rightarrow_{CBCP} & [\![(\neg P), (\neg Q), (\neg R), \mathbf{f}]\!] \quad \text{(2nd rule, () = } \mathbf{f} \text{)} \\
\Rightarrow_{CBCP} & [\![\mathbf{f}]\!] \quad \text{(1st rule)}
\end{array}$$

2.5.3 Formula BCP and Prime BCP

Formula BCP [McA80, McA90] extends Clausal BCP to any propositional theory. Given any theory Γ, Formula BCP monotonically expands it by adding facts as follows: in each step, if any *single* formula in Γ and *all the facts* in Γ taken together *logically entail* any other fact, then the new fact is added to the theory Γ. This step is repeated until no new fact can be so obtained.

Since Formula BCP is identical to Clausal BCP for clausal theories, it is also incomplete. Moreover, the satisfiability problem (SAT) can be trivially reduced to the general problem of determining whether a fact is logically entailed by an arbitrary (possibly, non-clausal) formula. Thus, the fact-inference problem for Formula BCP is CoNP-Hard, i.e., intractable.

Prime BCP [de 90] has been proposed as an algorithm for implementing Formula BCP. The basic idea is to first compute all the prime implicates[3] of each formula in the given theory, and then use Clausal BCP on the theory containing just these prime implicates. For any theory, Prime BCP infers the same facts as Formula BCP [de 90]: thus, Prime BCP is also incomplete, and the fact-inference problem for Prime BCP is also CoNP-Hard, i.e., intractable. Moreover, the number of prime implicates of a formula can be exponential in the size of the formula.

[3] The prime implicates of a theory are the minimal clauses that are logically entailed by the theory [BB70]

$$
\begin{array}{rcl}
\odot(\alpha, B_2) & \Rightarrow & \odot(\vee(\alpha), B_2) \\
\odot(\wedge(B_1), B_2) & \Rightarrow & \odot(B_1, B_2) \\
\vee(\vee(B_1), B_2) & \Rightarrow & \vee(B_1, B_2) \\
\vee(\alpha, \alpha, B) & \Rightarrow & \vee(\alpha, B) \\
\odot(\vee(\sim\alpha, \alpha, B_1), B_2) & \Rightarrow & \odot(B_2) \\
\vee(\wedge(\psi_0, \ldots, \psi_n), B_1) & \Rightarrow & \wedge(\vee(\psi_0, B_1), \ldots, \vee(\psi_n, B_1))
\end{array}
$$

where α's are literals, ψ's are formulas, and B's are bags of formulas.

Figure 2.7: A rewrite system for CNF transformation

2.5.4 CNF-BCP

CNF-BCP [de 90] infers facts by first transforming the given theory into conjunctive normal form (CNF) and then using Clausal BCP. We first provide a non-traditional definition of CNF transformation and then discuss CNF-BCP.

Intuitively, the CNF transformation of a theory produces a logically equivalent clausal theory using simple syntactic operations. For example, the CNF transformation of the theory $\odot(((P \wedge Q) \vee \neg R))$ produces the theory $\odot([\![\vee([\![P, \neg R]\!]), \vee([\![Q, \neg R]\!])]\!])$. The basic idea is to recursively combine the CNF transformations of the formulas in the theory, where CNF transformation of a formula is a bag of sets of literals (which intuitively represents a clausal theory) — the CNF transformation of a conjunctive formula is the union of CNF transformations of its conjuncts, whereas the CNF transformation of a disjunctive formula requires a kind of cross-product of CNF transformations of its disjuncts (in the base case, the CNF transformation of a literal is a singleton bag with a singleton set containing the literal). Representing CNF transformations by the function CNF, we obtain the following for the above example:

$$
\begin{array}{rcl}
\mathrm{CNF}(P) & = & [\![\{P\}]\!] \\
\mathrm{CNF}(Q) & = & [\![\{Q\}]\!] \\
\mathrm{CNF}(\wedge([\![P, Q]\!])) & = & [\![\{P\}, \{Q\}]\!] \\
\mathrm{CNF}(\neg R) & = & [\![\{\neg R\}]\!] \\
\mathrm{CNF}(\vee([\![\wedge([\![P, Q]\!]), \neg R]\!])) & = & [\![\{P, \neg R\}, \{Q, \neg R\}]\!]
\end{array}
$$

Definition 2.25 The cross-product $\biguplus$, which maps bags of bags of sets of literals to bags of sets of literals, is defined recursively as follows:

1. $\biguplus([\![]\!]) = [\![\emptyset]\!]$;
2. $\biguplus([\![R]\!] \cup B) = [\![L \cup S \mid L \in R, b \in \biguplus(B)]\!]$;

where L and S are sets of literals, R is a bag of sets of literals, and B is a bag of bags of sets of literals. The function $\underline{\mathrm{CNF}}$, which maps formulas to bags of sets of literals and theories to clausal theories, is defined recursively as follows:

1. for any literal α, $\mathrm{CNF}(\alpha) = [\![\{\alpha\}]\!]$;
2. for any bag B of formulas,
   (a) $\mathrm{CNF}(B) = [\![\mathrm{CNF}(\psi) \mid \psi \in B]\!]$ (hence $\mathrm{CNF}([\![]\!]) = [\![]\!]$);

(b) $\mathrm{CNF}(\wedge(B)) = \cup(\mathrm{CNF}(B))$ (hence $\mathrm{CNF}(\mathbf{t}) = \llbracket\rrbracket$);

(c) $\mathrm{CNF}(\vee(B)) = \llbracket L \in \biguplus(\mathrm{CNF}(B)) \mid L \cap \sim L = \emptyset \rrbracket$ (hence $\mathrm{CNF}(\mathbf{f}) = \llbracket\emptyset\rrbracket$);

(d) $\mathrm{CNF}(\odot(B)) = \odot(\llbracket \vee(L) \mid L \in \cup(\mathrm{CNF}(B))\rrbracket)$
(hence $\mathrm{CNF}(\odot()) = \llbracket\rrbracket$, $\mathrm{CNF}(\odot(\mathbf{t})) = \llbracket\rrbracket$, and $\mathrm{CNF}(\odot(\mathbf{f})) = \llbracket\mathbf{f}\rrbracket$);

where $\sim L = \{\sim a \mid a \in L\}$ for any set L of literals. ∎

Note that $\biguplus(\llbracket R \rrbracket) = \llbracket R \rrbracket$ for any bag R of sets of literals. Also, the condition $L \cap \sim L = \emptyset$ in the definition of $\mathrm{CNF}(\vee(B))$ is used intuitively to filter out sets containing complementary literals, for example, the set $\{P, \neg P, R\}$. Further, $\mathrm{CNF}(B)$ for a bag B of formulas is used merely as a short-hand notation in defining $\mathrm{CNF}(\wedge(B))$, $\mathrm{CNF}(\vee(B))$, and $\mathrm{CNF}(\odot(B))$. In the above example,

$$\begin{aligned}
\mathrm{CNF}(\llbracket P, Q \rrbracket) &= \llbracket \llbracket\{P\}\rrbracket, \llbracket\{Q\}\rrbracket \rrbracket \\
\mathrm{CNF}(\llbracket \wedge(\llbracket P, Q \rrbracket), \neg R \rrbracket) &= \llbracket \llbracket\{P\}, \{Q\}\rrbracket, \llbracket\{\neg R\}\rrbracket \rrbracket
\end{aligned}$$

The definition of $\mathrm{CNF}(\odot(B))$ is identical to that of $\mathrm{CNF}(\wedge(B))$, except that the result is a clausal theory.

Incidentally, CNF transformation can also be defined using the rewrite system of Figure 2.7 (this result is not used in this thesis). Using this rewrite system, the theory Γ can be reduced to $\mathrm{CNF}(\Gamma)$ in a single rewrite using the last rule:

$$\llbracket((P \wedge Q) \vee \neg R)\rrbracket \Rightarrow_R \llbracket(P \vee \neg R), (Q \vee \neg R)\rrbracket$$

CNF-BCP is strictly weaker than Formula BCP. For example, only Formula BCP obtains P from the theory $\odot(((P \vee Q) \wedge (P \vee \neg Q)))$. The algorithm CNF-BCP has exponential time complexity in the worst case, since the CNF transformation itself may lead to an exponential increase in the size of the theory. CNF transformations may also spoil the natural structure of theories (for example, locality [de 90]). Since there is no known PTIME algorithm for inferring the facts specified by CNF-BCP, the fact-inference problem for CNF-BCP is not known to be tractable.

2.5.5 Discussion

BCP algorithms perform two distinct tasks: inferring facts (literals) from a theory, and simplifying a theory. The two tasks are related: simplification proceeds by first inferring facts, and more facts may be inferred after simplification. However, the complexities of the two problems (not the BCP algorithms) may differ. For example, although the full simplified theory in CNF-BCP may be exponential in the size of the input theory, it is not known whether the fact-inference problem for CNF-BCP is either NP-Hard or CoNP-Hard.

Since the fact-inference problem for Formula BCP (and Prime BCP) is intractable (CoNP-Hard), we restrict our attention to CNF-BCP for the rest of this paper.

Definition 2.26 For any theory Γ, $\mathrm{BCP}(\Gamma)$ is the unique irreducible theory such that $\mathrm{CNF}(\Gamma) \Rightarrow^!_{CBCP} \mathrm{BCP}(\Gamma)$. Any clause in any theory Δ such that $\mathrm{CNF}(\Gamma) \Rightarrow^*_{CBCP} \Delta$ is said to be *produced by BCP* from theory Γ. ∎

Note that all variants of BCP, except Horn Pebbling, that are discussed in this section produce the same facts for clausal theories.

2.6 FPC : Extending Clausal BCP

Fact Propagation among Clauses (FPC) extends Clausal BCP to be applicable to every theory in PCE. The basic idea is to split the second rewrite rule of Clausal BCP (Figure 2.6) into two distinct steps:

$$
\begin{array}{llll}
S1_\wedge & \wedge(\mathbf{f}, B) \Rightarrow \wedge(\mathbf{f}) & S2_\wedge & \wedge(\mathbf{t}, B) \Rightarrow \wedge(B) \\
S1_\vee & \vee(\mathbf{t}, B) \Rightarrow \vee(\mathbf{t}) & S2_\vee & \vee(\mathbf{f}, B) \Rightarrow \vee(B) \\
S1_\odot & \odot(\mathbf{f}, B) \Rightarrow \odot(\mathbf{f}) & S2_\odot & \odot(\mathbf{t}, B) \Rightarrow \odot(B) \\
S3_\wedge & \wedge(\psi) \Rightarrow \psi & S3_\vee & \vee(\psi) \Rightarrow \psi
\end{array}
$$

where ψ is a formula and B is a bag of formulas; B must be non-empty in $S1$ rules.

Figure 2.8: Simplification rules

1. replace $\sim\alpha$ in $\vee(\sim\alpha, B)$ by $\mathbf{f}$, and
2. replace $\vee(\mathbf{f}, B)$ by $\vee(B)$.

The first step suggests that literals may be propagated through the rest of the theory by substituting logical constants for them. The second step suggests that theories can be simplified by eliminating these logical constants. We observe that these steps are applicable in a more general setting, and construct general rules out of them. After presenting the rewrite system FPC, we will prove that it is convergent, monotonic, content preserving, and modular. We also show that FPC is at least as powerful as BCP for clausal theories. Later, in Section 3.4, we will present a tractable algorithm for FPC, while in Section 2.7, we will extend FPC by adding more rules.

FPC is a rewrite system consisting of two kinds of rules:

Simplification Rules: These are the rules $S1\_$,$S2\_$, and $S3\_$ of Figure 2.8. They simplify terms containing $\mathbf{t}$, $\mathbf{f}$, and redundant connectives. For example, $\odot(\vee(\mathbf{t}, P)) \Rightarrow_R \odot(\vee(\mathbf{t}))$ using the rule $\vee(\mathbf{t}, B) \Rightarrow \vee(\mathbf{t})$ denoted by $S1_\vee$. To ensure termination, the bag B in $S1$ rules must be non-empty; otherwise, we obtain infinite non-terminating sequences of rewriting, such as $\odot(\mathbf{f}) \Rightarrow_R \odot(\mathbf{f}) \Rightarrow_R \ldots$

Propagation Rules: These are the $P1$ rules of Figure 2.9. They propagate a literal α by replacing each occurrence of α and $\sim\alpha$ in B by some logical constant. Again, the atom of α must be a subterm of B to ensure termination.

For example, the formula $(Q \vee (P \wedge (\neg P \vee Q)))$ can be reduced in the following sequence:

$$
\begin{array}{lll}
(Q \vee (P \wedge (\neg P \vee Q))) & \Rightarrow_{FPC} & (Q \vee (P \wedge (\mathbf{f} \vee Q))) \quad (\text{rule } P1_\wedge) \\
 & \Rightarrow_{FPC} & (Q \vee (P \wedge \vee(Q))) \quad (\text{rule } S2_\vee) \\
 & \Rightarrow_{FPC} & (Q \vee (P \wedge Q)) \quad (\text{rule } S3_\vee) \\
 & \Rightarrow_{FPC} & (Q \vee (P \wedge \mathbf{f})) \quad (\text{rule } P1_\vee) \\
 & \Rightarrow_{FPC} & (Q \vee \wedge(\mathbf{f})) \quad (\text{rule } S1_\wedge) \\
 & \Rightarrow_{FPC} & (Q \vee \mathbf{f}) \quad (\text{rule } S3_\wedge) \\
 & \Rightarrow_{FPC} & \vee(Q) \quad (\text{rule } S2_\vee) \\
 & \Rightarrow_{FPC} & Q \quad (\text{rule } S3_\vee)
\end{array}
$$

Thus, we can reduce $(Q \vee (P \wedge (\neg P \vee Q)))$ to Q. Since no rule applies to Q, it is a normal form of $(Q \vee (P \wedge (\neg P \vee Q)))$. A different reduction sequence for the same formula is:

$$
\begin{array}{ll}
P1_{\wedge} & \wedge(\alpha, B) \Rightarrow \wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]) \\
P1_{\vee} & \vee(\alpha, B) \Rightarrow \vee(\alpha, B[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]) \\
P1_{\odot} & \odot(\alpha, B) \Rightarrow \odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])
\end{array}
$$

where α is a literal and B is a bag of formulas such that atom of α is a subterm of B.

Figure 2.9: Propagation rules

$$
\begin{array}{lll}
(Q \vee (P \wedge (\neg P \vee Q))) & \Rightarrow_{FPC} & (Q \vee (P \wedge (\neg P \vee \mathbf{f}))) \quad (\text{rule } P1_{\vee}) \\
& \Rightarrow_{FPC} & (Q \vee (P \wedge \vee(\neg P))) \quad (\text{rule } S2_{\vee}) \\
& \Rightarrow_{FPC} & (Q \vee (P \wedge \neg P)) \quad (\text{rule } S3_{\vee}) \\
& \Rightarrow_{FPC} & (Q \vee (\neg P \wedge \mathbf{f})) \quad (\text{rule } P1_{\wedge}) \\
& \Rightarrow_{FPC} & (Q \vee \wedge(\mathbf{f})) \quad (\text{rule } S1_{\wedge}) \\
& \Rightarrow_{FPC} & (Q \vee \mathbf{f}) \quad (\text{rule } S3_{\wedge}) \\
& \Rightarrow_{FPC} & \vee(Q) \quad (\text{rule } S2_{\vee}) \\
& \Rightarrow_{FPC} & Q \quad (\text{rule } S3_{\vee})
\end{array}
$$

Even with this different sequence, we obtain the same irreducible formula Q. Also, note that Q is logically equivalent to $(Q \vee (P \wedge (\neg P \vee Q))$.

Note that the various rule schemas are grouped together using the following conventions (where Ri is either $S1$, $S2$, $S3$, or $P1$):

1. lhs of $Ri_{\wedge}$ is a conjunctive formula, while rhs is a formula;
2. lhs of $Ri_{\vee}$ is a disjunctive formula, while rhs is a formula;
3. both lhs and rhs of $Ri_{\odot}$ are theories;
4. $Ri_{\wedge}$ and $Ri_{\vee}$ are duals of each other;
5. $Ri_{\wedge}$ is the cdual of $Ri_{\odot}$.

We will follow this grouping convention for all rewrite systems for fact propagation.

2.6.1 Properties of FPC

If follows directly from the grouping of schemas that the rewrite system FPC is closed with respect to duals and cduals. Theorem 2.17 proves that it is convergent, content preserving, monotonic, and modular. For this, we use the results presented in Section 2.4.3, and the following lemmas.

Lemma 2.12 *$l \equiv r$ for each rewrite rule $l \Rightarrow r$ in FPC.*

Proof: We show that $v(l) = v(r)$ for any interpretation v and any rule $l \Rightarrow r$ in FPC:

$S1$**:** both lhs and rhs of $S1_{\wedge}$ and $S1_{\odot}$ are *false* in v, whereas both lhs and rhs of $S1_{\vee}$ are *true* in v.

$S2$: For $S2_\wedge$:

$$\begin{array}{lll} v(\wedge(\mathbf{t}, B)) = \mathit{true} & \text{iff} & v(\psi) = \mathit{true} \text{ for each } \psi \in B \\ & \text{iff} & v(\wedge(B)) = \mathit{true} \end{array}$$

For $S2_\vee$, replace $\wedge$, $\mathbf{t}$, and *true* above by their respective duals. For $S2_\odot$, replace $\wedge$ by $\odot$.

$S3$: $v(\psi) = \mathit{true}$ iff $v(\wedge(\psi)) = \mathit{true}$ iff $v(\vee(\psi)) = \mathit{true}$.

$P1$: For $P1_\wedge$:

$$\begin{array}{lll} v(\wedge(\alpha, B)) = \mathit{true} & \text{iff} & v(\alpha) = \mathit{true} \text{ and } v(\wedge(B)) = \mathit{true} \\ & \text{iff} & v(\alpha) = \mathit{true} \text{ and } v(\wedge(B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])) = \mathit{true} \\ & \text{iff} & v(\wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])) = \mathit{true} \end{array}$$

For $P1_\vee$, replace $\wedge$, $\mathbf{t}$, and *true* above by their respective duals. For $P1_\odot$, replace $\wedge$ by $\odot$.

■

Lemma 2.13 *$l \succ r$ for each rewrite rule $l \Rightarrow r$ in FPC.*

Proof:

$S1$: Since B is not empty, r is obtained by removing at least one formula from l. Thus, $w_1(l) \geq w_1(r)$ and $w_2(l) \geq w_2(r)$, and at least one of them is a strict inequality, i.e., $l \succ r$.

$S2$: r is obtained by removing a single logical constant from l. Thus $w_1(l) = w_1(r)$ and $w_2(l) = w_2(r) + 1$, i.e., $l \succ r$.

$S3$: r is obtained by removing a single connective from l. Thus $w_1(l) = w_1(r)$ and $w_2(l) = w_2(r) + 1$, i.e., $l \succ r$.

$P1$: $w_1(l) > w_1(r)$ since r is obtained by removing at least one literal from l. Thus, $l \succ r$.

■

Lemma 2.14 $\mathrm{facts}(l) \subseteq \mathrm{facts}(r)$ *for each rewrite rule $l \Rightarrow r$ in FPC.*

Proof: Note that $\{\mathbf{t}\} \subseteq \mathrm{facts}(r)$ for any logical term (formula or theory) r. For each of the rules $S1_\wedge, S1_\vee, S2_\wedge, P1_\wedge, P1_\vee$, $\mathrm{facts}(l) = \{\mathbf{t}\}$. For each of the rules $S1_\odot, S2_\odot, S3_\wedge, S3_\vee$, $\mathrm{facts}(l) = \mathrm{facts}(r)$. For rule $S2_\vee$, if $B = [\![]\!]$ then $\mathrm{facts}(l) = \mathrm{facts}(r)$; otherwise $\mathrm{facts}(l) = \{\mathbf{t}\}$. For rule $P1_\odot$, if $\sim\alpha \in \mathrm{facts}(\odot(B))$ then $\mathrm{facts}(r)$ is the set of all facts; otherwise $\mathrm{facts}(l) = \mathrm{facts}(r)$. ■

Lemma 2.15 $\odot(B', B_1) \Leftrightarrow^*_{FPC} \odot(B', B_2)$ *for each rewrite rule $\odot(B_1) \Rightarrow \odot(B_2)$ in FPC and each bag B' of formulas.*

Proof: We show that, in each case, both $\odot(B', B_1)$ and $\odot(B', B_2)$ rewrite to the same theory:

$S1_\odot$: If $\odot(\mathbf{f}, B) \Rightarrow \odot(\mathbf{f})$ is an instance of $S1_\odot$, then both $\odot(\mathbf{f}, B, B')$ and $\odot(\mathbf{f}, B')$ reduce to $\odot(\mathbf{f})$, each using at most one application of $S1_\odot$.

$S2_\odot$: If $\odot(\mathbf{t}, B) \Rightarrow \odot(B)$ is an instance of $S2_\odot$, then $\odot(\mathbf{t}, B, B')$ rewrites to $\odot(B, B')$ using $S2_\odot$.

$P1_\odot$: If $\odot(\alpha, B) \Rightarrow \odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ is an instance of $P1_\odot$, then B must contain at least one occurrence of α or its complement; hence the condition for rule $P1_\odot$ is satisfied for $\odot(\alpha, B, B')$, which rewrites to $\odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ using $P1_\odot$. On the other hand, $\odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B')$ equals $\odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ if B' does not contain α, or rewrites to the same theory using a second application of $P1_\odot$, if B' does contain α.

■

Lemma 2.16 *Each directed pair of rule schemas in FPC is confluent.*

Proof: Appendix D shows this explicitly for certain pairs. The claim then follows from Propositions 2.8, 2.9, and 2.10. ■

Combining all these results, we obtain the main result of this section:

Theorem 2.17 *The rewrite system FPC is convergent, content preserving, monotonic, and modular.*

Proof: Termination of FPC follows from Lemmas 2.5 and 2.13 and Proposition 2.6. Confluence (and hence, convergence) of FPC then follows from Lemmas 2.16 and 2.7. FPC is content preserving, using Lemmas 2.2 and 2.12. Monotonicity of FPC follows from Lemmas 2.3 and 2.14. Modularity of FPC follows from Lemmas 2.4 and 2.15. ■

The next theorem shows that the irreducible form of any clausal theory with respect to FPC is the same as that obtained by Clausal BCP.

Theorem 2.18 *For any clausal theory Γ, $\Gamma \Rightarrow^{!}_{FPC}$* BCP($\Gamma$).

Proof: For any clausal theory Γ, we know that $\Gamma \Rightarrow^{!}_{CBCP}$ BCP(Γ), where CBCP is the rewrite system given in Figure 2.6. Since it can be easily verified that BCP(Γ) is irreducible with respect to FPC, it suffices to show that that $l \Leftrightarrow^{*}_{FPC} r$ for each rewrite rule $l \Rightarrow r$ of CBCP. Since the first rule is already in FPC, we need to show this only for the second rule:

$$\begin{array}{rll}
l = \odot(\vee(\alpha), \vee(\sim\alpha, B_1), B_2) & \Rightarrow_{FPC} & \odot(\alpha, \vee(\sim\alpha, B_1), B_2) \quad (\text{rule } S3_{\vee}) \\
& \Rightarrow_{FPC} & \odot(\alpha, \vee(\mathbf{f}, B_1[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]) \quad (\text{rule } P1_{\odot}) \\
& \Rightarrow_{FPC} & \odot(\alpha, \vee(B_1[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]) \quad (\text{rule } S2_{\vee}) \\
r = \odot(\vee(\alpha), \vee(B_1), B_2) & \Rightarrow_{FPC} & \odot(\alpha, \vee(B_1), B_2) \quad (\text{rule } S3_{\vee}) \\
& \Rightarrow^{*}_{FPC} & \odot(\alpha, \vee(B_1[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]) \quad (\text{rule } P1_{\odot})
\end{array}$$

Since both l and r reduce to the same formula, $l \Leftrightarrow^{*}_{FPC} r$. ■

Note that Clausal BCP is not defined for non-clausal theories, but FPC is also applicable to non-clausal theories. However, FPC infers fewer facts than CNF-BCP for some theories, for example, the theory $\Gamma_1 = [\![((P \wedge Q) \vee (P \wedge \neg Q))]\!]$. Since Γ_1 is irreducible with respect to to FPC, no facts are inferred using FPC. On the other hand, the CNF transformation of Γ_1 produces the theory $[\![P, (P \vee Q), (P \vee \neg Q)]\!]$, from which BCP infers the fact P. Intuitively, CNF transformation allows the atom P to be factored out, since it occurs in all the disjuncts. In the next section, we present a rewrite system that has such factoring capabilities.

2.7 FP : Extending CNF-BCP

We now present a rewrite system FP that can be used to infer more facts than CNF-BCP, and prove that it is convergent, modular, content preserving, and monotonic. FP is obtained by adding rewrite rules to FPC that allow factoring of common literals from subformulas, as is possible using the CNF transformation (see the discussion of Γ_1 at the end of the previous section). In order to maintain confluence and modularity we need to add several additional, rather specialized, rules. Because of these restrictions, some rules that apply to conjunctive formulas might not apply to theories. (A tractable algorithm for FP will be presented in Section 3.6.)

$$
\begin{array}{ll}
L1_\wedge & \wedge(\wedge(\alpha, B_1), B_2) \Rightarrow \wedge(\alpha, \wedge(B_1), B_2) \\
L1_\vee & \vee(\vee(\alpha, B_1), B_2) \Rightarrow \vee(\alpha, \vee(B_1), B_2) \\
L1_\odot & \odot(\wedge(\alpha, B_1), B_2) \Rightarrow \odot(\alpha, \wedge(B_1), B_2) \\
L2_\wedge & \wedge(\alpha_1, \ldots, \alpha_n, \wedge(B)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, B) \\
L2_\vee & \vee(\alpha_1, \ldots, \alpha_n, \vee(B)) \Rightarrow \vee(\alpha_1, \ldots, \alpha_n, B)
\end{array}
$$

where $n \in \mathcal{N}$, α's are literals, and B's are bags of formulas.

Figure 2.10: Lifting rules

Recall from the previous section that while CNF-BCP factors out P from the theory $[\![((P \wedge Q) \vee (P \wedge \neg Q))]\!]$, FPC does not infer P from this. In order to strengthen FPC to obtain this factoring, we can add the following rewrite rule:

$$\text{basic factoring} : \wedge(\vee(\alpha, B_0), \ldots, \vee(\alpha, B_m)) \Rightarrow \vee(\alpha, \wedge(\vee(B_0), \ldots, \vee(B_m)))$$

along with its dual and theory counterparts, where α is a literal and B's are bags of formulas. (Here we introduce our final notational convention for rule schemata: as with inference rule schemas in natural deduction, dots ... represent an arbitrary number of different terms of the same kind as the first one, but with all subscripted meta-variables distinct. Hence meta-rules are instantiated in two steps: first, a decision is made about the "dots" replacement, and then meta-variables are instantiated by terms.) It is clear that Γ_1 can be rewritten to $[\![(P \wedge (Q \vee \neg Q))]\!]$ using the dual of the basic factoring rule, and can then be reduced to $[\![P]\!]$ using FPC.

However, the resulting rewrite system is not confluent. For example, in the following theory:

$$\Gamma_2 = [\![((P \wedge Q \wedge R) \vee (P \wedge Q \wedge S))]\!]$$

we get the following distinct irreducible forms depending on whether we first factor P or Q:

$$[\![(P \wedge (Q \wedge (R \vee S)))]\!]$$
$$[\![(Q \wedge (P \wedge (R \vee S)))]\!]$$

In order to make it confluent again, we can add the following rewrite rule:

$$\text{Collapsing: } \wedge(\wedge(B_1), B_2) \Rightarrow \wedge(B_1, B_2)$$

along with its dual and theory counterparts, where B's are bags of formulas. It is clear that both the above theories can be reduced to the same theory

$$[\![(P \wedge Q \wedge (R \vee S))]\!]$$

using the new rule. Another advantage of using this rule is that facts in B_1 can now be propagated in the entire formula $\wedge(B_1, B_2)$, rather than just in $\wedge(B_1)$.

However, the resulting system is still not confluent. For example, in the following theory:

$$\Gamma_3 = [\![(((P \wedge Q) \vee (P \wedge R)) \vee S)]\!]$$

we get the following distinct irreducible forms depending on whether we first use factoring or collapsing:

$$[\![((P \wedge (Q \vee R)) \vee S)]\!]$$
$$[\![((P \wedge Q) \vee (P \wedge R) \vee S)]\!]$$

$$F1_\wedge$$
$$\wedge(\alpha_1,\ldots,\alpha_n,\vee(\alpha,B_0),\ldots,\vee(\alpha,B_m)) \;\Rightarrow\; \wedge(\alpha_1,\ldots,\alpha_n,\vee(\alpha,\wedge(\vee(B_0),\ldots,\vee(B_m))))$$
$$F1_\vee$$
$$\vee(\alpha_1,\ldots,\alpha_n,\wedge(\alpha,B_0),\ldots,\wedge(\alpha,B_m)) \;\Rightarrow\; \vee(\alpha_1,\ldots,\alpha_n,\wedge(\alpha,\vee(\wedge(B_0),\ldots,\wedge(B_m))))$$

where $n, m \in \mathcal{N}$ $(m > 1)$, α's are literals, and B's are bags of formulas.

Figure 2.11: Factoring rules

In other words, collapsing can block factoring.

In order to avoid such blocking, the collapsing rule can be replaced by the following weaker rule:

$$\text{Lifting-1}: \wedge(\wedge(\alpha,B_1),B_2) \;\Rightarrow\; \wedge(\alpha,\wedge(B_1),B_2)$$

along with its dual and theory counterparts, where α is a literal and B's are bags of formulas. The resulting system contains rules of FPC, basic factoring, and lifting. It can be verified that Γ_1 reduces to $[\![P]\!]$ and that Γ_2 and Γ_3 are reduced to the following normal forms, respectively:

$$[\![(P \wedge Q \wedge (R \vee S))]\!]$$
$$[\![((P \wedge (Q \vee R)) \vee S)]\!]$$

However, the resulting system is still not confluent because of the interaction with simplification rules. For example, in the following theory:

$$\Gamma_4 = [\![((P \wedge \mathbf{f}) \vee (P \wedge (Q \vee R) \wedge (Q \vee R)))]\!]$$

we get the following distinct irreducible forms depending on whether we first use $S1$ or factoring:

$$[\![(P \wedge (Q \vee R) \wedge (Q \vee R))]\!]$$
$$[\![(P \wedge (Q \vee R))]\!]$$

In order to reduce the former theory to the latter, we need to generalize basic factoring to the following rule (Factoring-1):

$$\wedge(\alpha_1,\ldots,\alpha_n,\vee(\alpha,B_0),\ldots,\vee(\alpha,B_m)) \;\Rightarrow\; \wedge(\alpha_1,\ldots,\alpha_n,\vee(\alpha,\wedge(\vee(B_0),\ldots,\vee(B_m))))$$

along with its dual and theory counterparts, where α's are literals, B's are bags of formulas, and $m \geq 1$. The resulting system contains rules of FPC, factoring, and lifting. It can be verified that Γ_4 reduces to the following normal form:

$$[\![(P \wedge (Q \vee R))]\!]$$

However, the resulting system is still not confluent. For example, in the following formula:

$$\wedge(\wedge(P,B))$$

where B is a bag containing at least two formulas such that $\wedge(B)$ is irreducible, we get the following distinct irreducible forms depending on whether we first use $S3$ or lifting:

$$\wedge(P,B)$$
$$\wedge(P,\wedge(B))$$

For another example, in the following formula:

$$\vee(\wedge(P,\mathbf{f}),\wedge(P,B))$$

where B is a bag containing at least two formulas such that $\wedge(B)$ is irreducible, we get the same two distinct irreducible forms given above, depending on whether we first use $S1$ or factoring.

In order to reduce the latter theory to the former, we need to add another lifting rule:

$$\text{Lifting-2}: \wedge(\alpha_1, \ldots, \alpha_n, \wedge(B)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, B)$$

along with its dual and theory counterparts. The resulting system contains rules of FPC, factoring rules, and both the lifting rules.

However, the resulting system is not modular. For example, while the theory

$$[\![\vee(P, B_1), \vee(P, B_2)]\!]$$

can be reduced to

$$[\![\vee(P, \wedge(\vee(B_1), \vee(B_2)))]\!]$$

using the factoring rule, the following two theories are not reducible to the same theory:

$$[\![B, \vee(P, B_1), \vee(P, B_2)]\!]$$
$$[\![B, \vee(P, \wedge(\vee(B_1), \vee(B_2)))]\!]$$

The problem is that factoring applicable to a subtheory may not be applicable to the entire theory. One way to achieve modularity would be to detect all potential factorings that apply to any subtheory of a theory. Since the number of different sub-theories is exponential in the number of formulas in a theory, this will cause intractability. So, we pursue another alternative for retaining modularity: restrict factoring so that it does not apply at all to theories or sub-theories. As a consequence, although P would be inferable from the theory $[\![((P \vee Q) \wedge (P \vee R))]\!]$ because factoring applies to formulas, P is not inferable from the theory $[\![(P \vee Q), (P \vee R)]\!]$.

However, with this change, the resulting system is not confluent. For example, in the following theory:

$$[\![P, \wedge(\vee(Q, B_1), \vee(Q, B_2))]\!]$$

we get the following distinct irreducible forms depending on whether we first use factoring or the second lifting rule:

$$[\![P, \vee(Q, \wedge(\vee(B_1), \vee(B_2)))]\!]$$
$$[\![P, \vee(Q, B_1), \vee(Q, B_2)]\!]$$

In order to retain confluence, we should restrict the second lifting rule so that it does not apply to theories. The resulting system contains rules of FPC, factoring rules, and both the lifting rules. We will show that this system, which we will call Fact Propagation, FP, given in Figure 2.12, is convergent, content preserving, monotonic, and modular. Note that there are no theory counterparts of $L2$ and $F1$ rules; thus, theories are reduced differently from conjunctions involving the same formula.

The following is an example of a reduction sequence for the theory $\odot(\vee(\wedge(P, Q, R), \wedge(P, \neg Q, R)), \wedge(R, \vee(S, Q)))$ using the rules in FP:

$$\begin{array}{ll}
\Rightarrow_{FP} & \odot(\vee(\wedge(P, Q, R), \wedge(P, \neg Q, R)), R, \wedge(\vee(S, Q))) \quad (\text{rule } L1_{\odot}) \\
\Rightarrow_{FP} & \odot(\vee(\wedge(P, Q, \mathbf{t}), \wedge(P, \neg Q, \mathbf{t})), R, \wedge(\vee(S, Q))) \quad (\text{rule } P1_{\odot}) \\
\Rightarrow^*_{FP} & \odot(\vee(\wedge(P, Q), \wedge(P, \neg Q)), R, \wedge(\vee(S, Q))) \quad (\text{rule } S2_{\wedge}) \\
\Rightarrow_{FP} & \odot(\vee(\wedge(P, Q), \wedge(P, \neg Q)), R, \vee(S, Q)) \quad (\text{rule } S3_{\wedge}) \\
\Rightarrow_{FP} & \odot(\vee(\wedge(P, \vee(\wedge(Q), \wedge(\neg Q)))), R, \vee(S, Q)) \quad (\text{rule } F1_{\vee}) \\
\Rightarrow_{FP} & \odot(\wedge(P, \vee(\wedge(Q), \wedge(\neg Q))), R, \vee(S, Q)) \quad (\text{rule } S3_{\vee}) \\
\Rightarrow^*_{FP} & \odot(\wedge(P, \vee(Q, \neg Q)), R, \vee(S, Q)) \quad (\text{rule } S3_{\wedge})
\end{array}$$

Simplification Rules:

$$
\begin{array}{llll}
S1_{\wedge} & \wedge(\mathbf{f}, B) \Rightarrow \wedge(\mathbf{f}) & S2_{\wedge} & \wedge(\mathbf{t}, B) \Rightarrow \wedge(B) \\
S1_{\vee} & \vee(\mathbf{t}, B) \Rightarrow \vee(\mathbf{t}) & S2_{\vee} & \vee(\mathbf{f}, B) \Rightarrow \vee(B) \\
S1_{\odot} & \odot(\mathbf{f}, B) \Rightarrow \odot(\mathbf{f}) & S2_{\odot} & \odot(\mathbf{t}, B) \Rightarrow \odot(B) \\
S3_{\wedge} & \wedge(\psi) \Rightarrow \psi & S3_{\vee} & \vee(\psi) \Rightarrow \psi
\end{array}
$$

Propagation Rules:

$$
\begin{array}{ll}
P1_{\wedge} & \wedge(\alpha, B) \Rightarrow \wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]) \\
P1_{\vee} & \vee(\alpha, B) \Rightarrow \vee(\alpha, B[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]) \\
P1_{\odot} & \odot(\alpha, B) \Rightarrow \odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])
\end{array}
$$

Lifting Rules:

$$
\begin{array}{ll}
L1_{\wedge} & \wedge(\wedge(\alpha, B_1), B_2) \Rightarrow \wedge(\alpha, \wedge(B_1), B_2) \\
L1_{\vee} & \vee(\vee(\alpha, B_1), B_2) \Rightarrow \vee(\alpha, \vee(B_1), B_2) \\
L1_{\odot} & \odot(\wedge(\alpha, B_1), B_2) \Rightarrow \odot(\alpha, \wedge(B_1), B_2) \\
L2_{\wedge} & \wedge(\alpha_1, \ldots, \alpha_n, \wedge(B)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, B) \\
L2_{\vee} & \vee(\alpha_1, \ldots, \alpha_n, \vee(B)) \Rightarrow \vee(\alpha_1, \ldots, \alpha_n, B)
\end{array}
$$

Factoring Rules:

$$
\begin{array}{c}
F1_{\wedge} \\
\wedge(\alpha_1, \ldots, \alpha_n, \vee(\alpha, B_0), \ldots, \vee(\alpha, B_m)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, \vee(\alpha, \wedge(\vee(B_0), \ldots, \vee(B_m)))) \\
F1_{\vee} \\
\vee(\alpha_1, \ldots, \alpha_n, \wedge(\alpha, B_0), \ldots, \wedge(\alpha, B_m)) \Rightarrow \vee(\alpha_1, \ldots, \alpha_n, \wedge(\alpha, \vee(\wedge(B_0), \ldots, \wedge(B_m))))
\end{array}
$$

where $n, m \in \mathcal{N}$, α's are literals, ψ is a formula, and B's are bags of formulas. To ensure termination, B can't be an empty bag in $S1$ rules, the atom of α must be a subterm of B in $P1$ rules, and $m \geq 1$ in $F1$ rules.

Figure 2.12: Rewrite system FP

$$\begin{aligned}
&\Rightarrow_{FP} \quad \odot(\wedge(P, \vee(Q, \mathbf{t})), R, \vee(S, Q)) \quad (\text{rule } P1_{\vee}) \\
&\Rightarrow_{FP} \quad \odot(\wedge(P, \vee(\mathbf{t})), R, \vee(S, Q)) \quad (\text{rule } S1_{\vee}) \\
&\Rightarrow_{FP} \quad \odot(\wedge(P, \mathbf{t}), R, \vee(S, Q)) \quad (\text{rule } S3_{\vee}) \\
&\Rightarrow_{FP} \quad \odot(\wedge(P), R, \vee(S, Q)) \quad (\text{rule } S2_{\wedge}) \\
&\Rightarrow_{FP} \quad \odot(P, R, \vee(S, Q)) \quad (\text{rule } S3_{\wedge})
\end{aligned}$$

The rewrite system FP doesn't always produce the logically "simplest" formula. For example, consider the formulas $((P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q))$ and $(P \wedge Q)$. Although the two are logically equivalent, each of them is irreducible. In fact, it is not possible to have a tractable rewrite system (unless P = NP) that produces the logically "simplest" formula, since it can then be used to determine satisfiability.

2.7.1 Properties of FP

If follows directly from the grouping of schemas that the rewrite system FP is closed with respect to dual and cdual. We now prove that it is convergent, content preserving, monotonic, and modular. For this, we use the results presented in Section 2.4.3.

First, we show that FP is identical to FPC for clausal theories:

Proposition 2.19 *For any clausal theory Γ and any theory Δ in PCE, $\Gamma \Leftrightarrow^*_{FP} \Delta$ iff $\Gamma \Leftrightarrow^*_{FPC} \Delta$.*

Proof: None of the rules of FP that is not in FPC can be used to rewrite a theory that does not have either a conjunctive formula or a formula with nested disjunctions. Since a clausal theory never rewrites to any such theory using a rule in FP, the claim follows. ■

Lemma 2.20 *$l \equiv r$ for each rewrite rule $l \Rightarrow r$ in FP.*

Proof: It follows from Lemma 2.12 that all we need to show is that $l \equiv r$ for any rewrite rule $l \Rightarrow r$ of FP that is not in FPC. The result is obvious for lifting rules. For factoring rules (see Figure 2.12):

$F1_{\wedge}$: For any interpretation v:

$$\begin{aligned}
v(l) = true \quad &\text{iff} \quad v(\alpha_1) = \ldots = v(\alpha_n) = true, \text{ and} \\
&\phantom{\text{iff}} \quad \text{for each } i = 1 \ldots m, \text{ either } v(\alpha) = true \text{ or } v(\vee(B_i)) = true \\
&\text{iff} \quad v(\alpha_1) = \ldots = v(\alpha_n) = true, \text{ and} \\
&\phantom{\text{iff}} \quad \text{either } v(\alpha) = true \text{ or} \\
&\phantom{\text{iff}} \quad \text{for each } i = 1 \ldots m\text{: } v(\vee(B_i)) = true \\
&\text{iff} \quad v(r) = true
\end{aligned}$$

$F1_{\vee}$: Replace $\wedge$, $\vee$, and $true$ above by their respective duals.

■

Lemma 2.21 *$l \succ r$ for each rewrite rule $l \Rightarrow r$ in FP.*

Proof: It follows from Lemma 2.13 that all we need to show is that $l \succ r$ for each rewrite rule $l \Rightarrow r$ of FP that is not in FPC:

$L1$: Since r is obtained by moving a literal out of a connective in l, $w_3(l) > w_3(r)$. Since, $w_1(l) = w_1(r)$ and $w_2(l) = w_2(r)$, it follows that $l \succ r$. Note that this holds even if the bag B_1 is empty.

$L2$: Since r is obtained by removing a connective in l, $w_2(l) > w_2(r)$. Since $w_1(l) = w_1(r)$, it follows that $l \succ r$. Note that this holds even if the bag B is empty or if $n = 0$.

$F1$: Since $m \geq 1$, r is obtained by removing at least one literal in l. Thus, $w_1(l) > w_1(r)$, i.e., $l \succ r$.

■

If $m = 0$ were allowed in $F1$ rules, then it would be possible to obtain a cycle of reductions:

$$\begin{array}{lll} \wedge(\vee(\alpha, B)) & \Rightarrow_R & \wedge(\vee(\alpha, \wedge(\vee(B)))) \quad \text{(unrestricted rule } F1_\wedge) \\ & \Rightarrow_R & \wedge(\vee(\alpha, \vee(B))) \quad \text{(rule } S3_\wedge) \\ & \Rightarrow_R & \wedge(\vee(\alpha, B)) \quad \text{(rule } L2_\vee) \end{array}$$

Lemma 2.22 facts$(l) \subseteq$ facts(r) *for each rewrite rule* $l \Rightarrow r$ *in* FP.

Proof: It follows from Lemma 2.14 that all we need to show is that facts$(l) \subseteq$ facts(r) for each rewrite rule $l \Rightarrow r$ of FP that is not in FPC. For either $L1_\wedge$ or $L1_\vee$, or any of the factoring rules, facts$(l) = \{\mathbf{t}\}$. For rule $L1_\odot$, if $B_1 = [\![]\!]$ then facts$(l) =$ facts(r); otherwise facts$(l) =$ facts$(\odot(B_2)) \subseteq$ facts(r). For $L2$ rules, if $n \leq 1$ and $B = [\![]\!]$ then facts$(l) =$ facts(r); otherwise facts$(l) = \{\mathbf{t}\}$. ■

For the above proof to go through the $L1_\vee$ and $L1_\wedge$ cases, it was important that facts$(\wedge(\wedge(P))) = \{\mathbf{t}\}$.

Lemma 2.23 $\odot(B', B_1) \Leftrightarrow^*_{FP} \odot(B', B_2)$ *for any rewrite rule* $\odot(B_1) \Rightarrow \odot(B_2)$ *in* FP *and any bag* B' *of formulas.*

Proof: It follows from Lemma 2.15 that we need to prove the claim only for the rule $L1_\odot$. This follows directly since $\odot(\wedge(\alpha, B_1), B_2, B') \Rightarrow_{FP} \odot(\alpha, \wedge(B_1), B_2, B')$ using the rule $L1_\odot$. ■

Lemma 2.24 *Each directed pair of rule schemas in FP is confluent.*

Proof: Appendix D shows this explicitly for certain pairs. The claim then follows from Lemma 2.16 and Propositions 2.8, 2.9, and 2.10. ■

Combining all these results, we obtain the main result of this section:

Theorem 2.25 *The rewrite system* FP *is convergent, content preserving, monotonic, and modular.*

Proof: Termination follows from Lemmas 2.5 and 2.21 and Proposition 2.6. Confluence (and hence, convergence) then follows from Lemmas 2.24 and 2.7. FP is content preserving, using Lemmas 2.2 and 2.20. Monotonicity follows from Lemmas 2.3 and 2.22. Modularity follows from Lemmas 2.4 and 2.23. ■

Since FP is convergent, it follows that the reduction relation $\Rightarrow^!_{FP}$ is a function. We will denote this function by FPF. In other words, for any theory Γ in PCE without equality, $\underline{\text{FPF}(\Gamma)}$ is the unique irreducible theory such that $\Gamma \Rightarrow^!_{FP}$ FPF(Γ). In section 2.8, we will extend FPF to all the theories in PCE.

2.7.2 Comparison with CNF-BCP

For the purpose of this section, we restrict our attention to PCE without equality, a syntactic variant of finite PC. Thus, there is no equality predicate in formulas and theories, all the atoms are propositions, and all theories are finite.

We will prove that FP infers more facts than CNF-BCP. This will follow as a corollary of a theorem which states that for any clause $\vee(\alpha_0, \ldots, \alpha_n)$ $(n \geq 0)$ produced by CNF-BCP on any theory Γ, the fact α_n is inferred by FP from the theory $\Gamma \cup [\![\sim\alpha_0, \ldots, \sim\alpha_{n-1}]\!]$. The theorem is first proved for the case when ψ is a clause in CNF(Γ) itself, i.e., even before Clausal BCP is used. This weaker result, which is proved by induction on the construction of Γ, is based on two lemmas that show that a fact can be inferred by FP from a conjunctive (or a disjunctive) formula if it can be inferred by FP from some conjunct (each disjunct, respectively) in the formula. Both of these lemmas are based on the result that if a fact α is inferred using FP by propagating some literals through a formula ψ then FP produces the formula $\wedge(\alpha, B)$ (for some bag B of formulas) by rewriting the formula obtained from ψ by replacing each occurrence of those literals by $\mathbf{t}$. To prove these claims, we will extensively use the properties of FP given in Theorem 2.25.

We first show some simple properties about FP that will be used later in the section. First, the terms $\odot(\mathbf{f})$ and $\odot(\alpha)$ for any literal α are irreducible with respect to FP. The next proposition shows that facts directly inferable from any theory irreducible with respect to FP are present explicitly as formulas in the theory:

Proposition 2.26 *For any theory Γ irreducible with respect to FP and any literal α, $\alpha \in \mathrm{facts}(\Gamma)$ iff either $\alpha \in \Gamma$ or $\Gamma = \odot(\mathbf{f})$.*

Proof: By definition of facts, if $\Gamma = \odot(\mathbf{f})$ then $\alpha \in \mathrm{facts}(\Gamma)$. Now $\alpha \in \mathrm{facts}(\Gamma)$ and $\Gamma \neq \odot(\mathbf{f})$ iff either α or $\wedge(\alpha)$ or $\vee(\alpha)$ in Γ iff $\alpha \in \Gamma$ (since the other two formulas are reducible). This proves the claim. ■

The next proposition shows some special properties of the theory $[\![\mathbf{f}]\!]$:

Proposition 2.27 *For any theory Γ irreducible with respect to FP and any atom p: $\Gamma = \odot(\mathbf{f})$ iff $\mathbf{f} \in \mathrm{facts}(\Gamma)$ iff $\{p, \neg p\} \subseteq \mathrm{facts}(\Gamma)$.*

Proof:

1. if $\Gamma = \odot(\mathbf{f})$ then $\mathbf{f} \in \mathrm{facts}(\Gamma)$ follows directly from the definition of facts.
2. if $\mathbf{f} \in \mathrm{facts}(\Gamma)$ then $\mathrm{facts}(\Gamma)$ contains all facts (from the definition of facts). Thus, $\{p, \neg p\} \subseteq \mathrm{facts}(\Gamma)$.
3. suppose $\{p, \neg p\} \subseteq \mathrm{facts}(\Gamma)$ for some atom p, but $\Gamma \neq \odot(\mathbf{f})$. There are only three possibilities, each of which will lead to a contradiction since Γ is no longer irreducible:
   (a) either $\wedge(\mathbf{f})$ or $\vee(\mathbf{f})$ in Γ
   (b) one of $\wedge(p)$, $\wedge(\neg p)$, $\vee(p)$, or $\vee(\neg p)$ is in Γ
   (c) both p and $\neg p$ are in Γ.

■

The next proposition shows that more facts can be inferred using FP from larger theories:

Proposition 2.28 *For any theories Γ and Δ: $\mathrm{facts}(\mathrm{FPF}(\Gamma)) \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma \cup \Delta))$.*

Proof: Since FP is modular, $\mathrm{FPF}(\Gamma \cup \Delta) = \mathrm{FPF}(\mathrm{FPF}(\Gamma) \cup \Delta)$ using Lemma 2.1. By convergence of FP, $\mathrm{FPF}(\Gamma) \cup \Delta \Rightarrow^*_{FP} \mathrm{FPF}(\Gamma \cup \Delta)$. The claim then follows from the monotonicity of FP. ■

We now prove that if a fact α is inferred using FP by propagating some literals through a formula ψ then FP produces the formula $\wedge(\alpha, B)$ (for some bag B of formulas) by rewriting the formula obtained from ψ by replacing each occurrence of those literals by $\mathbf{t}$. We also consider the special case when $\mathbf{f}$ is so inferred. We first have to define this notion of replacement:

Definition 2.27 Any bag A of literals is <u>*consistent*</u> iff there is no atom p such that both p and $\neg p$ are in A. For any term t and any consistent bag $A = [\![\alpha_1, \ldots, \alpha_n]\!]$ of literals, $t[\mathbf{t} \hookleftarrow A]$ is defined to be the term $t[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha_1] \ldots [\mathbf{t} \overset{*\sim}{\leftarrow} \alpha_n]$. ■

Note that the term $t[\mathbf{t} \hookleftarrow A]$ does not depend on the ordering of literals in A, since A is consistent and all literals in A are replaced by the same term $\mathbf{t}$.

Lemma 2.29 *For any consistent bag A of literals, any formula ψ, and any literal α not in A:*

1. *if* $\mathrm{FPF}(\odot(A,\psi)) = \odot(\mathbf{f})$ *then* $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \mathbf{f}$*;*
2. *if* $\alpha \in \mathrm{FPF}(\odot(A,\psi))$ *then either* $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \alpha$ *or there is a nonempty bag B of formulas such that* $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \wedge(\alpha, B)$.

Proof: Using rule $P1_{\odot}$, we obtain $\odot(A,\psi) \Rightarrow^{*}_{FP} \odot(A, \psi[\mathbf{t} \hookleftarrow A])$.

1. Suppose $\mathrm{FPF}(\odot(A,\psi)) = \odot(\mathbf{f})$. Since FP is confluent, $\odot(A, \psi[\mathbf{t} \hookleftarrow A]) \Rightarrow^{*}_{FP} \odot(\mathbf{f})$. Since A is consistent and none of the atoms in A occurs in $\psi[\mathbf{t} \hookleftarrow A]$, the only way this rewriting can happen is when $\mathbf{f}$ is obtainable as a formula by reducing $\psi[\mathbf{t} \hookleftarrow A]$. Thus, $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \mathbf{f}$.
2. Suppose $\alpha \in \mathrm{FPF}(\odot(A,\psi))$. Since FP is confluent, we obtain that $\alpha \in \mathrm{FPF}(\odot(A, \psi[\mathbf{t} \hookleftarrow A]))$. Since $\alpha \notin A$, arguing as above, α must be obtained by reducing $\psi[\mathbf{t} \hookleftarrow A]$. This can only happen when either $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \wedge(\alpha, B)$ (for some non-empty bag B) or $\psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \alpha$. In the former case, rule $L1_{\odot}$ can be then used for lifting α out of the conjunction.

■

The next two lemmas show that a fact can be inferred by FP from a conjunctive (or a disjunctive) formula if it can be inferred by FP from each conjunct (some disjunct, respectively) in the formula.

Lemma 2.30 *For any bag B of formulas, any bag A of literals, and any literal α: if* $\alpha \in \mathrm{facts}(\mathrm{FPF}(\odot(A,\psi)))$ *for each formula $\psi \in B$, then* $\alpha \in \mathrm{facts}(\mathrm{FPF}(\odot(\vee(B), A)))$.

Proof: Denote $\odot(\vee(B), A)$ by Γ. Note that $A \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma))$ follows from Lemma 2.1, since FPF is monotonic with respect to facts. We consider various possibilities. Due to Proposition 2.27, in some cases it is sufficient to show that $\mathrm{FPF}(\Gamma) = \odot(\mathbf{f})$.

1. if $B = [\![]\!]$ then $\Gamma \Rightarrow_{FP} \odot(\mathbf{f})$ using rule $S1_{\odot}$, since $\vee(B) = \mathbf{f}$.
2. otherwise, if $\alpha \in A$ then the claim again follows by monotonicity, since $A \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma))$.
3. otherwise, if A is inconsistent then $FPF(\Gamma) = \odot(\mathbf{f})$ from Proposition 2.27, since $A \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma))$.
4. otherwise, using Lemma 2.29, we can split B into the following pairwise disjoint bags B_1, B_2, and B_3:

$$\begin{aligned} B_1 &= [\![\psi \in B \mid \psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \mathbf{f}]\!] \\ B_2 &= [\![\psi \in B \mid \psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \alpha]\!] \\ B_3 &= [\![\psi \in B \mid \psi[\mathbf{t} \hookleftarrow A] \Rightarrow^{!}_{FP} \wedge(\alpha, B_\psi) \text{ for some bag } B_\psi]\!] \end{aligned}$$

   Using rewrite rule $P1_{\odot}$, $\Gamma \Rightarrow^{*}_{FP} \odot(A, \vee(B)[\mathbf{t} \hookleftarrow A])$. Thus, using the above split:

$$\Gamma \Rightarrow^{*}_{FP} \odot(A, \vee([\![\mathbf{f} \mid \psi \in B_1]\!], [\![\alpha \mid \psi \in B_2]\!], [\![\wedge(\alpha, B_\psi) \mid \psi \in B_3]\!]))$$

   Since FP is convergent, the claim follows in each possible case:

   (a) if B_2 is non-empty, then using simplification rules and $P1_{\vee}$, we obtain that $\Gamma \Rightarrow^{!}_{FP} \odot(A,\alpha) = \mathrm{FPF}(\Gamma)$. Thus, $\alpha \in \mathrm{facts}(\mathrm{FPF}(\Gamma))$.

   (b) otherwise, if both B_2 and B_3 are empty, then $\Gamma \Rightarrow^{!}_{FP} \odot(\mathbf{f})$ using simplification rules. Thus, the claim follows.

   (c) otherwise, if B_3 is a singleton, say $[\![\psi]\!]$, then using simplification rules and $L1_{\odot}$, we obtain that $\Gamma \Rightarrow^{*}_{FP} \odot(A, \alpha, \wedge(B_\psi))$. The claim follows from the monotonicity of FP with respect to facts.

(d) otherwise, using simplification rules, $L1_{\odot}$, and $F1_{\vee}$, we obtain:

$$\Gamma \Rightarrow^{*}_{FP} \odot(A, \alpha, \wedge(\vee(\wedge(B_{\psi} \mid \psi \in B_3))))$$

The claim again follows directly from the monotonicity of FP with respect to facts.

∎

Lemma 2.31 *For any bag B of formulas, any bag A of literals, and any literal α: if $\alpha \in \mathrm{facts}(\mathrm{FPF}(\odot(A, \psi)))$ for some formula $\psi \in B$, then $\alpha \in \mathrm{facts}(\mathrm{FPF}(\odot(\wedge(B), A)))$.*

Proof: Denote $\odot(\wedge(B), A)$ by Γ. Note that $A \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma))$, since FPF is monotonic with respect to facts. We consider various possibilities. (Due to Proposition 2.27, in some cases it will be sufficient to show that $\mathrm{FPF}(\Gamma) = \odot(\mathbf{f})$.)

1. if A is inconsistent, then $FPF(\Gamma) = \odot(\mathbf{f})$ from Proposition 2.27, since $A \subseteq \mathrm{facts}(\mathrm{FPF}(\Gamma))$.
2. otherwise, using rewrite rule $P1_{\odot}$, $\Gamma \Rightarrow^{*}_{FP} \odot(A, \wedge(B)[\mathbf{t} \hookleftarrow A])$. Splitting B into B_1, B_2, and B_3, as in the proof of Lemma 2.30, we obtain:

$$\Gamma \Rightarrow^{*}_{FP} \odot(A, \wedge([\![\mathbf{f} \mid \psi \in B_1]\!], [\![\alpha \mid \psi \in B_2]\!], [\![\wedge(\alpha, B_{\psi}) \mid \psi \in B_3]\!]))$$

   If B_1 is non-empty then $\Gamma \Rightarrow^{*}_{FP} \odot(\mathbf{f})$ using rule $S1_{\odot}$, and the claim follows. Otherwise, either B_2 or B_3 is non-empty, since B contains a formula ψ. Using rule $L1_{\odot}$ (and possibly $L1_{\wedge}$), we obtain $\Gamma \Rightarrow^{*}_{FP} \odot(A, \alpha, B')$ for some bag B'. The claim then follows from the monotonicity of FP with respect to facts.

∎

The next lemma shows that propagating the complement of any immediate subclause of any clause in $\mathrm{CNF}(\Gamma)$ produces the remaining literal using FP.

Lemma 2.32 *For any theory Γ, any positive n, and any literals $\alpha_1, \ldots, \alpha_n$: if $\vee(\alpha_1, \ldots, \alpha_n) \in \mathrm{CNF}(\Gamma)$ then $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Gamma \cup [\![\sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]))$; if $\mathbf{f} \in \mathrm{CNF}(\Gamma)$ (the case when $n = 0$), then $\mathrm{facts}(\mathrm{FPF}(\Gamma)) = \mathrm{facts}(\odot(\mathbf{f}))$*

Proof: [By induction on the structure of Γ] We will prove the claim by induction (outer) on the number of formulas in Γ. A base case, when $\Gamma = [\![]\!]$, is trivial since $\mathrm{CNF}(\Gamma) = [\![]\!]$. Another base case, when Γ has a single formula, will be proved by another induction (inner) on the structure of this formula. Denote the theory $\Gamma \cup [\![\sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]$ by Γ'. There are three base cases for the inner induction:

$\Gamma = [\![\mathbf{t}]\!]$: trivial, since $\mathrm{CNF}(\Gamma) = [\![]\!]$.

$\Gamma = [\![\mathbf{f}]\!]$: trivial, since $\mathrm{CNF}(\Gamma) = \mathrm{FPF}(\Gamma) = \odot(\mathbf{f})$.

$\Gamma = [\![\alpha]\!]$ for some literal α: since cnf always returns disjunctions, $\mathrm{CNF}(\Gamma) = \odot(\vee(\alpha))$ and $\mathrm{FPF}(\Gamma) = \odot(\alpha)$.

There are two inductive steps for the inner induction:

$\Gamma = [\![\wedge(B)]\!]$ where B is a non-empty bag of formulas: The inductive assumption is that the lemma holds for any theory $[\![\psi]\!]$ where $\psi \in B$. Consider any $A = \vee(\alpha_1, \ldots, \alpha_n) \in \mathrm{CNF}(\Gamma)$. Thus, there must be a $\psi \in B$ such that $A \in \mathrm{CNF}([\![\psi]\!])$. Using the inductive assumption, $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Delta))$, where $\Delta = [\![\psi, \sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]$. Our claim follows directly from Lemma 2.31.

$\Gamma = [\![\vee(B)]\!]$ where B is a non-empty bag of formulas: The inductive assumption is that the lemma holds for any theory $[\![\psi]\!]$ where $\psi \in B$. Consider any $A = \{\alpha_1, \ldots, \alpha_n\}$ such that $\vee(A) \in \mathrm{CNF}(\Gamma)$. Thus, for each $\psi \in B$, there must be a set $A_\psi \subseteq A$ such that $A_\psi \in \mathrm{CNF}(\psi)$.

Consider any $\psi \in B$ and any $A' \in \mathrm{CNF}(\psi)$ such that $A' \subseteq A$. If A' is empty, then $\vee() = \mathbf{f} \in \mathrm{CNF}(\psi)$, so $\alpha_n \in \mathrm{facts}(\mathrm{FPF}([\![\psi]\!])) = \mathrm{facts}(\mathrm{FPF}(\Delta))$. Otherwise, let $A' = \{\delta_1, \ldots, \delta_p\}$, $\Delta' = [\![\psi, \sim\delta_1, \ldots, \sim\delta_{p-1}]\!]$, and $\Delta = [\![\psi, \sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]$. Using the inductive assumption, $\delta_p \in \mathrm{facts}(\mathrm{FPF}(\Delta'))$. Since $\Delta' \subseteq \Delta$, $\delta_p \in \mathrm{facts}(\mathrm{FPF}(\Delta))$, using Proposition 2.28. There are two subcases:

$\alpha_n \in A'$: Without loss of any generality, assume $\delta_p = \alpha_n$.

otherwise: Since $\sim\delta_p \in \Delta$, it follows from Proposition 2.27 that $\mathrm{FPF}(\Delta) = \odot(\mathbf{f})$.

Thus, $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Delta))$ is both cases.

Since this holds for each $\psi \in B$, our claim follows directly from Lemma 2.30.

We now consider the inductive step for the outer induction, i.e., Γ has at least two formulas. The inductive assumption is that the lemma holds for any theory $[\![\psi]\!]$ where $\psi \in \Gamma$. Consider any $A = \vee(\alpha_1, \ldots, \alpha_n) \in \mathrm{CNF}(\Gamma)$. There must be a $\psi \in \Gamma$ such that $A \in \mathrm{CNF}(\psi)$. Using the inductive assumption, $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Delta))$, where $\Delta = [\![\psi, \sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]$. Since $\Delta \subset \Gamma'$, our claim follows from Proposition 2.28. ∎

We are now ready to prove the main theorem of this section, which strengthens the above lemma for any clause produced by BCP on $\mathrm{CNF}(\Gamma)$.

Theorem 2.33 *For any theory Γ, any n, and any literals $\alpha_1, \ldots, \alpha_n$: if the clause $\vee(\alpha_1, \ldots, \alpha_n)$ is produced by BCP on $\mathrm{CNF}(\Gamma)$ then $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Gamma \cup [\![\sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]))$; if $\mathbf{f}$ is ever produced by BCP on $\mathrm{CNF}(\Gamma)$ (the claim for $n = 0$), then $\mathrm{FPF}(\Gamma \cup [\![\sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]) = \odot(\mathbf{f})$.*

Proof: Suppose $A = \vee(\alpha_1, \ldots, \alpha_n)$ is the first clause produced by BCP that violates the claim. Denote the theory $\Gamma \cup [\![\sim\alpha_1, \ldots, \sim\alpha_{n-1}]\!]$ by Γ'. Since it follows from Lemma 2.32 that $A \notin \mathrm{CNF}(\Gamma)$, A must have been produced by an application of the BCP inference rule. Thus, there is a literal α such that both $\vee(\sim\alpha)$ and $\vee(\alpha, \alpha_1, \ldots, \alpha_n)$ were earlier produced by BCP, and hence satisfy the claim. Thus, $\alpha \in \mathrm{facts}(\mathrm{FPF}(\Gamma))$ and $\alpha_n \in \mathrm{facts}(\mathrm{FPF}(\Delta))$, where $\Delta = \Gamma' \cup \{\sim\alpha\}$. Using Proposition 2.26, we obtain:

1. either $\mathrm{FPF}(\Gamma) = \odot(\mathbf{f})$ or $\sim\alpha \in \mathrm{FPF}(\Gamma)$, and
2. either $\mathrm{FPF}(\Delta) = \odot(\mathbf{f})$ or $\alpha_n \in \mathrm{FPF}(\Delta)$.

If $\mathrm{FPF}(\Gamma) = \odot(\mathbf{f})$ then $\mathrm{FPF}(\Gamma') = \odot(\mathbf{f})$ using Propositions 2.27 and 2.28, since $\Gamma \subseteq \Gamma'$. That is, A satisfies the claim, a contradiction. Thus, $\mathrm{FPF}(\Gamma) \neq \odot(\mathbf{f})$.

It follows from 1 that $\sim\alpha \in \mathrm{FPF}(\Gamma)$. Since $\Gamma \subseteq \Gamma'$, either $\mathrm{FPF}(\Gamma') = \odot(\mathbf{f})$ or $\sim\alpha \in \mathrm{FPF}(\Gamma')$, using Propositions 2.26 and 2.28. Using 2 and the modularity of FP, we obtain that either $\mathrm{FPF}(\Gamma') = \odot(\mathbf{f})$ or $\alpha_n \in \mathrm{FPF}(\Gamma')$; i.e., A satisfies the claim, a contradiction.

Thus, there is no clause A that violates the claim. ∎

A direct corollary of Theorem 2.33, when $n = 1$, shows that FP infers at least as many facts as CNF-BCP:

Corollary 2.34 *For any theory Γ: $\mathrm{facts}(BCP(\Gamma)) \subseteq \mathrm{facts}(FPF(\Gamma))$.*

Recall from Section 2.5 that for any theory Γ, $\mathrm{BCP}(\Gamma)$ denotes the theory obtained by CNF-BCP, i.e., first converting Γ to CNF and then using Clausal BCP.

The next example shows that FP may indeed infer more facts than BCP. Consider the theory $\Gamma = [\![(P \vee (Q \wedge (\neg Q \vee P)))]\!]$. It can be verified that $\mathrm{CNF}(\Gamma) = \mathrm{BCP}(\Gamma) = [\![(P \vee Q), (P \vee \neg Q)]\!]$; thus $\mathrm{facts}(\mathrm{BCP}(\Gamma)) = \{\mathbf{t}\}$. However, $\Gamma \Rightarrow^{!}_{FP} [\![P]\!]$; thus $\mathrm{facts}(\mathrm{BCP}(\Gamma)) \subset \{P, \mathbf{t}\} \subseteq \mathrm{facts}(FPF(\Gamma))$.

$$E1_\wedge \quad a \doteq a \;\Rightarrow\; \mathbf{t} \qquad\qquad E1_\vee \quad a \dot{\neq} a \;\Rightarrow\; \mathbf{f}$$

$$E2_\odot \quad \odot(a \doteq b, B) \;\Rightarrow\; \odot(a \doteq b, B[b \overset{*}{\leftarrow} a]) \qquad (\text{if } a \succ b)$$

where a and b are constants and B is a bag of formulas such that a is a subterm of B.

Figure 2.13: Equality rules

2.8 FPE : Handling Equality

We now extend FP to Fact Propagation with Equality, FPE, so as to be able to reason with simple cases of equality. The rewrite system FPE is proved to be convergent, modular, monotonic, and content preserving. As in the case of FP, we discard some alternative rewrite systems that do not satisfy these properties.

The simplest kind of equality reasoning is that the formula $(a \doteq a \vee P)$ should be reducible to $\mathbf{t}$. Also, the formula $(a \dot{\neq} a \wedge P)$ should be reducible to $\mathbf{f}$. The rewrite rules that will allow such reduction are:

$$\begin{array}{l} a \doteq a \;\Rightarrow\; \mathbf{t} \\ a \dot{\neq} a \;\Rightarrow\; \mathbf{f} \end{array}$$

We should also be able to reduce the theory $[\![a \doteq b, P(a), \neg P(b)]\!]$ to the theory $[\![\mathbf{f}]\!]$. The rewrite rule that will allow this is:

$$\odot(a \doteq b, B) \;\Rightarrow\; \odot(a \doteq b, B[b \overset{*}{\leftarrow} a]) \qquad (\text{if } a \succ b)$$

For this rule to be deterministic, we require that $a \succ b$. It is natural to also allow the counterparts of the above rule for conjunctive and disjunctive theories:

$$\begin{array}{l} \wedge(a \doteq b, B) \;\Rightarrow\; \wedge(a \doteq b, B[b \overset{*}{\leftarrow} a]) \\ \vee(a \dot{\neq} b, B) \;\Rightarrow\; \vee(a \dot{\neq} b, B[b \overset{*}{\leftarrow} a]) \end{array}$$

However, these rules may block applications of propagation and factoring rules, as the following two formulas demonstrate:

$$\begin{array}{c} (P(a,b) \wedge (a \neq b \vee P(a,b))) \Rightarrow_R (P(a,b) \wedge (a \neq b \vee P(b,b))) \\ ((a \doteq b \wedge P(b)) \vee (b \doteq c \wedge P(b))) \Rightarrow_R ((a \doteq b \wedge P(b)) \vee (b \doteq c \wedge P(c))) \end{array}$$

where $a \succ b \succ c$. In the former case, $P(a,b)$ can no longer be propagated using rule $P1_\wedge$; in the latter case, $P(b)$ can no longer be factored using rule $F1_\vee$. Confluence is violated because of this blocking.

Thus, we allow only those equality rules that are given in Figure 2.13. The resulting rewrite system, called FPE, is given in Figure 2.14.

For an example, suppose $a \succ b \succ c \succ d$:

$$\begin{array}{lll} [\![P(a,d), \neg P(c,b), a \doteq c, b \doteq d]\!] & \Rightarrow_{FPE} & [\![P(c,d), \neg P(c,b), a \doteq c, b \doteq d]\!] \quad (\text{rule } E2_\odot) \\ & \Rightarrow_{FPE} & [\![P(c,d), \neg P(c,d), a \doteq c, b \doteq d]\!] \quad (\text{rule } E2_\odot) \\ & \Rightarrow_{FPE} & [\![P(c,d), \mathbf{f}, a \doteq c, b \doteq d]\!] \quad (\text{rule } P1_\odot) \\ & \Rightarrow_{FPE} & [\![\mathbf{f}]\!] \quad (\text{rule } S1_\odot) \end{array}$$

2.8.1 Properties of FPE

If follows directly from the grouping of schemas that the rewrite system FPE is closed with respect to duals. Note that it is not closed with respect to cduals because of rule $E2_\odot$. We now prove that it is convergent, content preserving, monotonic, and modular. For this, we use the results presented in Section 2.4.3.

Simplification Rules:

$$S1_\wedge \quad \wedge(\mathbf{f}, B) \Rightarrow \wedge(\mathbf{f}) \qquad\qquad S2_\wedge \quad \wedge(\mathbf{t}, B) \Rightarrow \wedge(B)$$

$$S1_\vee \quad \vee(\mathbf{t}, B) \Rightarrow \vee(\mathbf{t}) \qquad\qquad S2_\vee \quad \vee(\mathbf{f}, B) \Rightarrow \vee(B)$$

$$S1_\odot \quad \odot(\mathbf{f}, B) \Rightarrow \odot(\mathbf{f}) \qquad\qquad S2_\odot \quad \odot(\mathbf{t}, B) \Rightarrow \odot(B)$$

$$S3_\wedge \quad \wedge(\psi) \Rightarrow \psi \qquad\qquad S3_\vee \quad \vee(\psi) \Rightarrow \psi$$

Propagation Rules:

$$P1_\wedge \quad \wedge(\alpha, B) \Rightarrow \wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$$

$$P1_\vee \quad \vee(\alpha, B) \Rightarrow \vee(\alpha, B[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha])$$

$$P1_\odot \quad \odot(\alpha, B) \Rightarrow \odot(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$$

Lifting Rules:

$$L1_\wedge \quad \wedge(\wedge(\alpha, B_1), B_2) \Rightarrow \wedge(\alpha, \wedge(B_1), B_2)$$

$$L1_\vee \quad \vee(\vee(\alpha, B_1), B_2) \Rightarrow \vee(\alpha, \vee(B_1), B_2)$$

$$L1_\odot \quad \odot(\wedge(\alpha, B_1), B_2) \Rightarrow \odot(\alpha, \wedge(B_1), B_2)$$

$$L2_\wedge \quad \wedge(\alpha_1, \ldots, \alpha_n, \wedge(B)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, B)$$

$$L2_\vee \quad \vee(\alpha_1, \ldots, \alpha_n, \vee(B)) \Rightarrow \vee(\alpha_1, \ldots, \alpha_n, B)$$

Factoring Rules:

$$F1_\wedge$$

$$\wedge(\alpha_1, \ldots, \alpha_n, \vee(\alpha, B_0), \ldots, \vee(\alpha, B_m)) \Rightarrow \wedge(\alpha_1, \ldots, \alpha_n, \vee(\alpha, \wedge(\vee(B_0), \ldots, \vee(B_m))))$$

$$F1_\vee$$

$$\vee(\alpha_1, \ldots, \alpha_n, \wedge(\alpha, B_0), \ldots, \wedge(\alpha, B_m)) \Rightarrow \vee(\alpha_1, \ldots, \alpha_n, \wedge(\alpha, \vee(\wedge(B_0), \ldots, \wedge(B_m))))$$

Equality Rules:

$$E1_\wedge \quad a \doteq a \Rightarrow \mathbf{t} \qquad\qquad E1_\vee \quad a \not\doteq a \Rightarrow \mathbf{f}$$

$$E2_\odot \quad \odot(a \doteq b, B) \Rightarrow \odot(a \doteq b, B[b \overset{*}{\leftarrow} a]) \qquad (\text{if } a \succ b)$$

where $n, m \in \mathcal{N}$, a and b are constants, α's are literals, ψ is a formula, and B's are bags of formulas. To ensure termination, B can't be an empty bag in $S1$ rules, the atom of α must be a subterm of B in $P1$ rules, $m \geq 1$ in $F1$ rules, and a must be a subterm of B in rule $E2$.

Figure 2.14: Rewrite system FPE used for fact propagation with equality

First, we observe that FPE is identical to FP for PCE without equality (finite PC), since no equality rules apply at any stage of rewriting.

Lemma 2.35 *$l \equiv r$ for each rewrite rule $l \Rightarrow r$ in FPE.*

Proof: It follows from Lemma 2.20 that all we need to show is that $l \equiv r$ for any rewrite rule $l \Rightarrow r$ of FPE that is not in FP. In the following, v is any interpretation:

$E1_\wedge$**:** $v(l) = true = v(r)$.

$E1_\vee$**:** $v(l) = false = v(r)$.

$E2_\odot$**:** Since interpretations must be consistent with equality:

$$\begin{aligned} v(\odot(a \doteq b, B)) = true \quad & \text{iff} \quad v(a \doteq b) = true \text{ and } v(\wedge(B)) = true \\ & \text{iff} \quad v(a \doteq b) = true \text{ and } v(\wedge(B[b \xleftarrow{*} a])) = true \\ & \text{iff} \quad v(\odot(a \doteq b, B[b \xleftarrow{*} a])) = true. \end{aligned}$$

■

Lemma 2.36 *$l \succ r$ for each rewrite rule $l \Rightarrow r$ in FPE.*

Proof: It follows from Lemma 2.21 that all we need to show is that $l \succ r$ for each rewrite rule $l \Rightarrow r$ of FPE that is not in FP:

$E1$**:** $w_1(l) > w_1(r)$ since r is obtained by replacing a literal in l by a logical constant. Thus, $l \succ r$.

$E2$**:** Since r is obtained by replacing at least one a in l by b and $a \succ b$, $w_4(l) \succ_{mul} w_4(r)$. Since, $w_1(l) = w_1(r)$, $w_2(l) = w_2(r)$, and $w_3(l) = w_3(r)$, it follows that $l \succ r$.

■

Lemma 2.37 facts$(l)^{\doteq} \subseteq$ facts$(r)^{\doteq}$ *for each rewrite rule $l \Rightarrow r$ in FPE.*

Proof: It follows from Lemma 2.22 that all we need to show is that facts$(l)^{\doteq} \subseteq$ facts$(r)^{\doteq}$ for each rewrite rule $l \Rightarrow r$ of FPE that is not in FP. For rules $E1_\wedge$ and $E2_\odot$, facts$(l)^{\doteq}$ = facts$(r)^{\doteq}$. For rule $E1_\vee$, facts(r) is the set of all facts. ■

Lemma 2.38 *$\odot(B', B_1) \Leftrightarrow^*_{FPE} \odot(B', B_2)$ for each rewrite rule $\odot(B_1) \Rightarrow \odot(B_2)$ in FPE and any bag B' of formulas.*

Proof: It follows from Lemma 2.23 that we need to prove the claim only for the rule $E2_\odot$. This follows directly, since both $\odot(a \doteq b, B, B')$ and $\odot(a \doteq b, B[b \xleftarrow{*} a], B')$ rewrite to $\odot(a \doteq b, B'[b \xleftarrow{*} a], B[b \xleftarrow{*} a])$ using zero or one application of rule $E2_\odot$. ■

Lemma 2.39 *Each directed pair of rule schemas in FPE is confluent.*

Proof: Appendix D verifies this explicitly for certain pairs. The claim then follows from Lemma 2.24 and Propositions 2.9, 2.8, and 2.10. Proposition 2.10 is applicable, since $E2_\odot$, the only theory rule that does not have a cdual, is not an extra rule used in proving confluence of any pair of rules. ■

Combining all these results, we obtain the main result of this section:

Theorem 2.40 *The rewrite system FPE is convergent, content preserving, monotonic, and modular.*

Proof: Termination of FPE follows from Lemmas 2.5 and 2.36 and Proposition 2.6. Confluence of FP then follows from Lemmas 2.39 and 2.7. FPE is content preserving, using Lemmas 2.2 and 2.35. Monotonicity of FPE follows from Lemmas 2.3 and 2.37. Modularity of FPE follows from Lemmas 2.4 and 2.38. ∎

Since FPE is convergent, the reduction relation $\Rightarrow^!_{FPE}$ is a function. Since $\Rightarrow^!_{FPE}$ agrees with $\Rightarrow^!_{FP}$ on theories without equality, we will use FPF to denote this function also. In other words, for any theory Γ in PCE, FPF(Γ) is the unique irreducible theory such that $\Gamma \Rightarrow^!_{FPE}$ FPF(Γ). In this way, FPF is defined for all theories in PCE.

2.9 Conclusions

We presented fact propagation, FP, a rewrite system for inferring facts from propositional theories. Though FP is logically incomplete, it does not require theories to be transformed into clausal form, which is an advantage in cases where such normalization causes an exponential increase in size or when explanations of the inferences need to be given to users. FP infers at least as many facts as inferred by CNF-BCP, which is an exponential time algorithm. For some theories, FP infers more facts than CNF-BCP.

We used rewrite systems, rather than inference systems, for defining fact propagation. There were several reasons for this. First, global changes are expressed conveniently using rewrite systems (for example, see propagation rules). Second, we do not need the old formulas after they have been simplified (for example, compare the inference rules and the rewrite system for clausal CNF). Third, as we shall see in Chapter 3, converting a rewrite system to an efficient algorithm can sometimes be easier, since the task at hand is well defined: find and maintain a list of remaining places where a rule can be applied.

Like other researchers dealing with rewrite systems, we found confluence to be a very important property. A confluent system results in a unique irreducible form for every term. The terminating nature of the rewriting system was helpful in proving the confluence of the system (because we only had to consider "local confluence" using single-step applications of pairs of overlapping rules). Confluence and termination also help in obtaining a tractable algorithm, as we shall see in the next chapter.

For these reasons, we used confluence and termination of the rewriting system as a heuristic for choosing some of the inferences our reasoner will perform (c.f. the section on FP). This was helpful because developers of incomplete reasoners traditionally face the difficult question of which inferences to make and which to avoid, and when to stop hunting for other inexpensive inferences to include.

Fact propagation does have limitations. First, it is quite weak in even some simple cases. For instance, it does not infer the fact P from the theory $[\![P \vee Q,\ P \vee \neg Q,\ Q \vee R]\!]$. Second, it lacks a model theory that provides an independent characterization of its reasoning.

Chapter 3

Algorithms for Fact Propagation

3.1 Overview

We present an algorithm, AFP, for computing the irreducible form of a finite theory using the rewrite system FP – a form that contains the facts inferred by FP. We show that algorithm AFP has quadratic time complexity in the number of propositional symbols and connectives in the input theory (the propositional symbols are assumed to be integers from 1 to k, where there are k distinct propositional symbols in the theory). If we restrict our attention to clausal theories, the complexity of AFP is the same as that of standard Clausal BCP: linear time.

AFP represents the input theory by a tree whose nodes are labeled by the subterms of the theory. It works by repeatedly rewriting the theory using some rewrite rule that is applicable to it. Rather than naively searching for an applicable rule in each step, which does not give a linear-time algorithm for clausal theories, AFP has several refinements:

- AFP uses data structures for efficiently identifying potential rule applications as soon as they become applicable. For example, instead of trying to apply a propagation rule by searching for each occurrence of a literal, AFP uses additional data structures (arrays called Occurs) for efficiently locating these occurrences without explicitly searching for them each time. Similarly, factoring rules use arrays Glits that keep count of the number of occurrences of the same literal nested in a specific way in the subformulas.
- Instead of re-initializing data structures each time the theory is modified (by replacing a subformula by a truth constant, say), some obsolete parts of the data structures are merely tagged and are modified only later when required (in a "lazy" fashion).
- Rather than applying a rule as soon as it becomes applicable, AFP keeps a record of potential rule applications in various queues. Rules are then applied in the following order until all the queues are emptied: propagation, lifting, and factoring; only simplification rules are applied as soon as they become applicable.

AFP uses the tree representation of a theory, as shown in Figure 2.1, where leaves are labeled by literals and internal nodes, which represent subformulas, are labeled by connectives. For each internal node N, each element N.Occurs[P] of an array called N.Occurs, indexed by literals, keeps a list of those nodes that provide access to all leaves in the subtree rooted at the N which are labeled by the literal P. Occurs arrays are used in efficiently applying propagation rules. An additional array N.Glits, indexed by integers, has the property that N.Glits[k] contains the list of literals that label k grandchildren of node N. Glits arrays are used to identify efficiently potential applications of factoring rules. Since initializing all entries of Occurs and Glits arrays is potentially expensive ($O(n^3)$), AFP initializes only those entries which correspond to the literals that actually appear in the corresponding subtree.

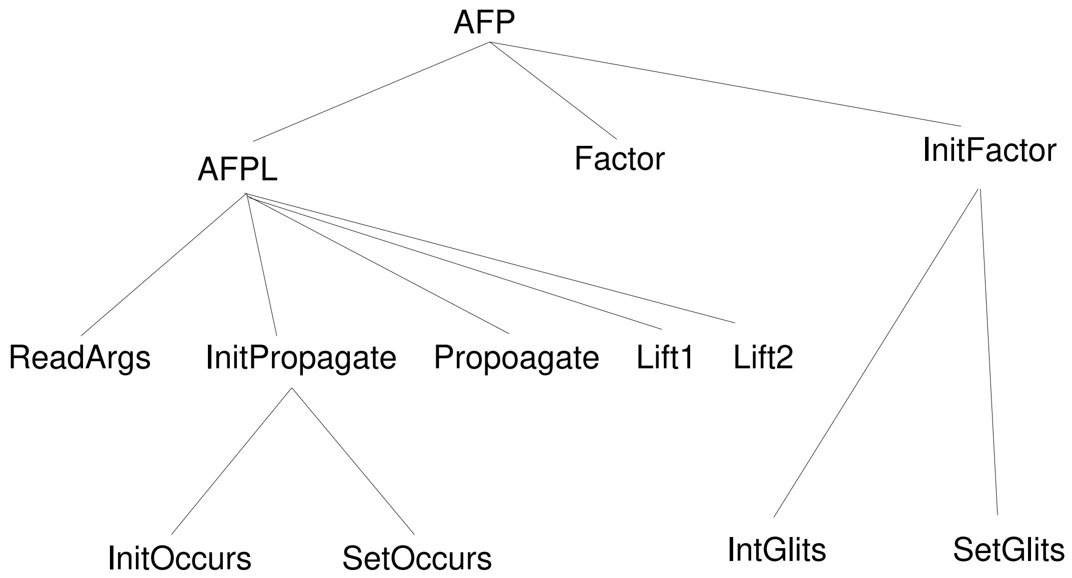

Figure 3.1: Nesting of procedures in algorithm AFP

Because of the complex data structures, the correctness of the AFP is not obvious. We therefore introduce several invariants about the data structures and show that the correctness of AFP follows from these invariants. Since the invariants are sometimes violated during the execution of the algorithm, we explicitly list all violations along with the culprit step in the algorithm, and show that each of these is rectified in a subsequent step.

To help the reader in understanding the AFP algorithm, we first present the algorithm AFPC which reduces an input theory using the rewrite system FPC, i.e., using only simplification and propagation rules. Without adding any new data structures (except for two queues of nodes), we then extend AFPC to the AFPL algorithm which also uses lifting rules. Finally, we present the algorithm AFP which also uses factoring rules. Even in AFP, the theory is first reduced with respect to simplification, propagation, and lifting rules. There are two main reasons for delaying the application of factoring rules, which are also the most complicated rules: factoring rules require additional data structures, whose initialization becomes much easier if the theory is irreducible with respect to the other rules; they are also the only rules that require adding new nodes to the tree, thus making it more difficult to analyze the complexity of the algorithm.

The following quick overview of the procedures to be described refers to Figure 3.1, which shows the nesting of procedures in the algorithm AFP. At the top level, AFP applies all possible rules except factoring (AFPL), initializes the data structures for factoring (InitFactor), and then applies all possible factoring rules and all other rules that become applicable. AFPL reads the input theory while applying all possible simplification rules and constructing its tree representation (ReadArgs), initializes the data structures for propagation and lifting rules (InitPropagate), and then applies all possible rules except factoring. For reducing the time-complexity, InitPropagate uses two passes over the entire tree: the first pass (InitOccurs) initializes all the relevant entries of the Occurs arrays to Nil, while the second pass (SetOccurs) sets them to the correct values. InitPropagate also creates a queue, PQ, of all nodes where propagation rule can be applied, i.e., the nodes that have at least one child labeled by a literal. Similarly, InitFactor uses two passes (InitGlits and SetGlits) for respectively initializing and setting the Glits arrays, and creates a queue, FQ, of all nodes where factoring rules can be applied. The queues PQ and FQ are updated whenever there are new possibilities for applying propagation and factoring rules, respectively. The two queues, L1Q and L2Q, for the two lifting rules are created from the nodes that are removed from the PQ list.

$$P2_\wedge \quad \wedge(\alpha, B) \Rightarrow \wedge(\alpha, B[\mathbf{t} \overset{\sim}{\leftarrow} \alpha])$$

$$P2_\vee \quad \vee(\alpha, B) \Rightarrow \vee(\alpha, B[\mathbf{t} \overset{\sim}{\leftarrow} {\sim}\alpha])$$

$$P2_\odot \quad \odot(\alpha, B) \Rightarrow \odot(\alpha, B[\mathbf{t} \overset{\sim}{\leftarrow} \alpha])$$

where α is a literal and B is a bag of formulas such that either α or ${\sim}\alpha$ is a subterm of B. Note that $B[\mathbf{t} \overset{\sim}{\leftarrow} \alpha]$ is obtained from B by replacing *some* occurrence of α by $\mathbf{t}$ or ${\sim}\alpha$ by $\mathbf{f}$.

Figure 3.2: Alternative Propagation Rules

Most of the rewriting of the input theory is done within the main loop of the algorithm. In each iteration, if PQ is non-empty then propagation rules are attempted (Propagate) at its first node; otherwise, if L1Q is non-empty then L1 rules are attempted (Lift1) at its first node; otherwise, if L2Q is non-empty then L2 rules are attempted (Lift2) at its first node; otherwise, if FQ is non-empty then factoring rules are attempted (Factor) at its first node. The algorithm terminates when either all the queues are empty or the theory reduces to $\{\mathbf{f}\}$ or $\{\ \}$. In all cases, the output theory is irreducible. Thus, the actual rewriting is done within procedures Propagate, Lift1, Lift2, and Factor; except for simplification rules which are also applied in ReadArgs. These procedures call several other procedures, for example, Collapse, which are explained later.

Applying simplification rules as soon as they become applicable causes a complication with propagation rules. Any application of a propagation rule may replace many literals by truth constants; our algorithms do these replacements in some arbitrary sequence. Since each of these replacements causes a new potential for simplification, many applications of simplification rules are mixed with one application of a propagation rule. Since this is not allowed in rewrite systems, we have to show that this mixing still gets the correct results. We do so by introducing a variant of propagation rules that replaces only one literal in each rule application, and proving that replacing the old propagation rule by this new one does not change the irreducible forms of terms.

We also extend AFP for dealing with the equality rules. The resulting algorithm AFPE has time complexity cubic in the size of the input theory. The increase in complexity is because of the need to change the atoms themselves due to replacement of constants in them.

The plan of the rest of this chapter is as follows: we first present an alternative version of propagation rules that is more easily implementable. We then present some basic data structures that are used throughout in AFP, and a technique to encode theories using sequences of integers which facilitates reading the input theory. This is followed by the four algorithms, AFPC, AFPL, AFP, and AFPE: for each of them, we first present the algorithm, argue its correctness using invariants, and then present an upper bound on its time complexity. These algorithms use several standard algorithmic “tricks” presented in the literature for reducing the time complexity (c.f. [AHU74]).

3.2 Alternative Propagation Rules

We introduce the new propagation rules that are implemented by our algorithms. We prove that they can be substituted (without changing the irreducible forms of terms) for the original propagation rules in any rewrite system whose termination is proved using the ordering $\succ$ defined in Section 2.4.3 . For the purposes of this section, we restrict our attention to only those rewrite systems that are terminating (see above) and contain all the original propagation rules.

The basic idea is to split each application of an original propagation rule (P1 rule) into a sequence of smaller steps, each of which may be followed by applications of S rules. These rules, $P2$, are given in

Figure 3.2. Here, $B[\mathbf{t} \overset{\sim}{\leftarrow} \alpha]$ is obtained from B by replacing some occurrence of α by $\mathbf{t}$ or $\sim\alpha$ by $\mathbf{f}$. The only difference from the $P1$ rules is that in the $P2$ rules, only one occurrence of α or $\sim\alpha$ is replaced by a truth constant, rather than all. Thus, there is a $P2$ rule corresponding to each $P1$ rule, and vice versa.

Definition 3.1 For any rewrite system R, its *P-alternative* rewrite system, R', is obtained by replacing each occurence of $\overset{*\sim}{\leftarrow}$ by $\overset{\sim}{\leftarrow}$ in the P1 rules of R. ∎

In order to show that a rewrite system R and its P-alternative system R' produce the same results, it suffices to prove the following claims: R' is terminating, R' is locally confluent iff R is, and the irreducible form of any term with respect to R is same as that with respect to R'. Recall that local confluence and termination guarantess confluence.

Lemma 3.1 *$l \succ r$ for any P2 rule $l \Rightarrow r$.*

Proof: Consider any P2 rule $l \Rightarrow r$. $w_1(l) > w_1(r)$ since r is obtained by removing exactly one literal from l. Thus, $l \succ r$. ∎

It follows directly from Lemmas 2.13 and 3.1 that the P-alternative system of any terminating rewrite system is also terminating. Recall that we restrict our attention to rewrite systems whose termination is proved using the ordering $\succ$ defined in Section 2.4.3.

The next lemma shows the relation between appplications of $P1$ and $P2$ rules. The subscript R there does not denote any particular rewrite system; it merely indicates that the lemma refers to rewrite rule applications, not the rules themselves — using our convention mentioned in Section 2.4.

Lemma 3.2 *For any terms s and t:*

1. *if $s \Rightarrow_R t$ using a P1 rule then $s \Rightarrow^*_{R'} t$ using P2 rules;*
2. *if $s \Rightarrow_{R'} t$ using a P2 rule then there is a term v such that $t \Rightarrow^*_{R'} v$ using P2 rules and $s \Rightarrow_R v$ using P1 rules.*

Proof:

1. Keep applying the corresponding P2 rule until it can be no longer applied to the same subterm for the same literal α.
2. v is obtained from t by repeatedly applying the same P2 rule until it cannot be applied. A single application of the corresponding P1 rule rewrites s to v.

∎

It follows directly from Claim 1 in Lemma 3.2 that for any rewrite system R and its P-alternative R', and any terms s and t, if $s \Rightarrow^*_R t$ then $s \Rightarrow^*_{R'} t$.

Lemma 3.3 *For any locally confluent rewrite system R, its P-alternative R' is locally confluent.*

Proof: For any terms s, t, x such that $s \Rightarrow_{R'} t$ and $s \Rightarrow_{R'} x$, we have to show that there is a term w such that $t \Rightarrow^*_{R'} w$ and $x \Rightarrow^*_{R'} w$. There are three distinct cases:

1. Both t and x are obtained from s using a rule other than P2. Since R is locally confluent, such a w exists.
2. Either t or x (but not both) is obtained from s using a P2 rule. Without loss of generality, assume that it is t. From Claim 2 of Lemma 3.2, there is a term v such that $t \Rightarrow^*_{R'} v$ using P2 rules and $s \Rightarrow_R v$ using P1 rules. Since R is locally confluent and $s \Rightarrow_R x$, there is a term w such that $v \Rightarrow^*_R w$ and $x \Rightarrow^*_R w$. Thus, $t \Rightarrow^*_{R'} w$ and $x \Rightarrow^*_{R'} w$.

3. Both t and x are obtained from s using P2 rules. From Claim 2 of Lemma 3.2, there is a term y such that $x \Rightarrow^*_{R'} y$ using P2 rules and $s \Rightarrow_R y$ using P1 rules. The argument given above, with x replaced by y, works in this case.

■

It follows directly from the conditions on rules $P1$ and $P2$ that a $P1$ rule applies to a term iff the corresponding $P2$ rule applies to that term. Thus, any term is irreducible with respect to a rewrite system iff it is irreducible with respect to its P-alternative system. Since local confluence and termination guarantees confluence, it follows directly from the above observation and Lemma 3.3 that the P-alternative rewrite system of any convergent rewrite system is also convergent. This brings to the main result of this section, which shows that the two rewrite systems also produce the same irreducible forms.

Theorem 3.4 *For any convergent rewrite system R, its P-alternative R', and any terms s and t: $s \Rightarrow^!_R t$ iff $s \Rightarrow^!_{R'} t$.*

Proof: (Only-if) Suppose $s \Rightarrow^!_R t$, i.e., $s \Rightarrow^*_R t$ and t is irreducible with respect to R. Thus, t is irreducible with respect to R', and it follows from Lemma 3.2 that $s \Rightarrow^*_{R'} t$. Thus, $s \Rightarrow^!_{R'} t$.

(If) Suppose $s \Rightarrow^!_{R'} t$. Since R is terminating, there is some term u such that $s \Rightarrow^!_R u$. It follows from the only-if direction of the theorem (proved above) that $s \Rightarrow^!_{R'} u$. Since R' is convergent, $u = t$. Thus, $s \Rightarrow^!_R t$.
■

Note that our goal in this chapter is to present a tractable algorithms for obtaining the irreducible forms of any given term with respect to the rewrite systems FPC, FP, and FPE of Chapter 2. It follows from Theorem 3.4 that our algorithms could be based instead on the P-alternatives of these rewrite systems. For the rest of this chapter, we will use these P-alternatives instead of the rewrite systems FPC, FP, and FPE. For simplicity, we continue to use the old names, for example, FP instead of FP$'$.

3.3 Basic Data Structures

We present some basic data structures used to represent a theory using a tree, some basic operations on those data structures, and some invariants that should hold at most times. Arguing that it is not necessary to have Occurs arrays for all the internal nodes of the tree, we introduce the notion of A-nodes (for any atom A) for which Occurs entries are required. We also define an encoding used for input theories.

As mentioned in Section 2.4, terms may be viewed as finite trees, the leaves of which are labeled with constants, 0-place predicates, and the empty bag, and the internal nodes of which are labeled with functions of positive arity, with out-degree equal to the arity of the label. For example, the theory $[\![P, ((\neg P \wedge Q) \vee P)]\!]$ is represented by the tree given in Figure 3.3. We will avoid representing **f** and **t** directly in trees — they will be immediately eliminated using S rules, so that the only nodes with 0 children will be leaves labelled by literals, and possibly the root.

A rewrite step using rule $l \Rightarrow r$ changes the subtree that represents the term l; we say that the rule is *applicable* to the root of the subtree. There is one exception to this: in case of L1 rules (for example, $L1_\wedge$), we say that the rule is applicable to the parent of the leaf labeled by α. The rewrite is also described by "applying the rule" at such a node. In general, we will use "tree" instead of the "term represented by the tree"; for example, "the tree is irreducible" rather than "the term represented by the tree is irreducible".

A tree is represented by maintaining the following information with each node (see Figures 3.3, 3.4, and 3.1, which are described later):

label: either a connective or a literal; note that after being created initially, the label of any node is never assigned to, and thus remains unchanged.

childs: (for any non-leaf node) list of children of the node; this list is composed of two disjoint parts: list **leafs** containing the leaf children and list **subs** containing the non-leaf children.

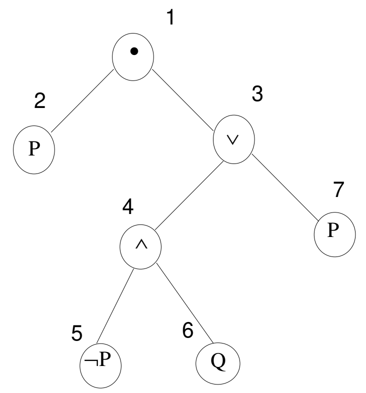

Figure 3.3: Tree representation of $[\![P, ((\neg P \wedge Q) \vee P)]\!]$

Conceptually, it is useful to associate unique names with different nodes — any reference to "list of nodes" is then understood as "list of names of nodes". In practise, it is not necessary to have explicit names, since there are other mechanisms (for example, pointers) for this purpose. Note that labels can not be used as names, since many nodes can have the same label.

For efficient processing, some additional (redundant) information is also explicitly maintained with each node (some of this will be made more clear as we go along):

parent: (for any non-root node N) parent of the node;

occurs: (for any non-leaf node N) array indexed by atoms, such that for any atom A, if subtrees rooted at at least two children of N have leaves labeled by A or $\neg A$, then N.occurs[A] is a list of descendents of N for accessing (through the occurs lists of those nodes recursively) all the leaves labeled by A or $\neg A$ in the subtree rooted at N.

pp: (for any leaf L), pointer to the root of some subtree in which it is known that there are no *other* occurrences of the same literal as that leaf (usually, the value of L.pp is L initially, and L.parent after propagation).

The "pp" field of a leaf is set to a node iff the leaf has been propagated in the subtree rooted at the node. This information is used in two ways:

1. a leaf is considered for propagation iff its "pp" field is not set to its parent;
2. the tree is irreducible with respect to propagation rules if the "pp" field of each leaf in the tree is set to its parent.

PP is initially set to the leaf itself, since the leaf has not been propagated in the subtree rooted at its parent.

Intuitively, the occurs list for a node N and an atom A encodes a tree rooted at N whose leaves are exactly those labeled by either A or $\neg A$. This list is used for efficiently applying propagation rules, during which the labels of these leaves are replaced by either **t** or **f**. Note that occurs lists are not required for all nodes of the tree, and that sometimes nodes deleted from the tree (by rules such as S3) will continue to appear in the occurs lists for the purposes of reaching appropriate leaf nodes.

Definition 3.2 For any trees T_1 and T_2, T_1 is a subset of T_2 if each node in T_1 is a node in T_2, and the ancestor relation in T_1 is a subset of that in T_2. For any atom A, an A-leaf is a leaf node that is labeled by either A or ¬A. For any atom A, an A-node is either the Root or a node that has *at least two* children whose

node	label	leafs	subs	parent	occurs[P]	occurs[Q]	pp
1	$\odot$	2	3		2,3	6	
2	P			1			2
3	$\vee$	7	4	1	5,7	Nil	
4	$\wedge$	5,6	Nil	3	Nil	Nil	
5	$\neg P$			4			5
6	Q			4			6
7	P			3			7

Table 3.1: Values of fields of nodes in the tree for $[\![P, ((\neg P \wedge Q) \vee P)]\!]$

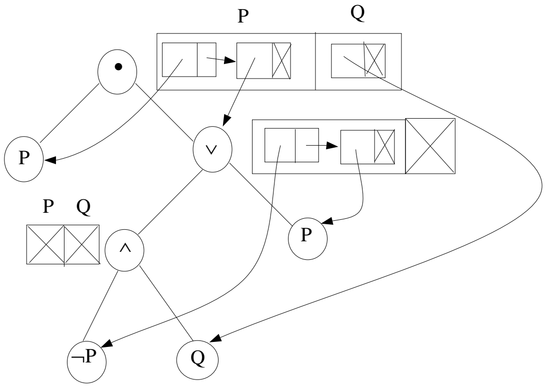

Figure 3.4: Occurs lists for the tree of $[\![P, ((\neg P \wedge Q) \vee P)]\!]$

trees have A-leaves. For any atom A, an A-tree is any subset of the tree rooted at Root which contains all A-leaves and all A-nodes in the tree rooted at Root. ∎

A propagation rule involving literal A or $\neg A$ needs to be applied at some node N iff N has such a literal as a child *and* N has at least one other occurrence of A or $\neg A$ under it. Therefore such propagation rules apply to node N iff it is an A-node, and hence we require occurs lists corresponding to A for only the A-nodes in the tree. In addition, the occurs lists are nested to avoid duplication — $N.occurs[A]$ may contain internal nodes also. In such cases, occurs lists of these nodes need to be recursively traversed to access all the A-leaves in the subtree.

Consider the tree given in Figure 3.3 that represents the theory $[\![P, ((\neg P \wedge Q) \vee P)]\!]$. The values of fields of the nodes in this tree are given in Table 3.1, where the nodes of the tree are numbered in preorder. The only P-nodes are 1 and 3, while there is no Q-node. The empty slots in the table indicate fields that are neither initialized nor used; note that since 1 is not a Q-node, its occurs[Q] list, though present, will never be used. The elements in the various occurs lists are also shown in Figure 3.4.

Some nodes and trees are considered special:

Definition 3.3 Tnode, Fnode are exceptional values, representing **t** and **f** during computations, but not appearing inside formula trees.

A Ttree is a tree containing a single node which is labeled by $\odot$. A Ftree is a tree whose root is labeled by $\odot$ and has some new, specially marked child. An exception tree is either Ttree or Ftree. ∎

There is a single exception condition for all the algorithms, namely, when the tree becomes an exception tree. In an exception, all computation stops and the corresponding exception tree is returned as the result, since theories represented by exception trees are irreducible. To maintain clarity, we will not mention this exception explicitly in the algorithms.

For proving the correctness of algorithms, we will establish and then maintain the following invariants in all trees, except exception trees:

Tree Invariant: For any node N, N is the root of the tree iff N.label = $\odot$; N is a leaf iff N.label is a literal; N is an internal node (i.e., neither root nor a leaf) iff N.label is either $\wedge$ or $\vee$, and N has at least two children.

Parent Invariant: For any pair of nodes N and P, N is in P.childs iff N.parent = P.

Occurs Invariant: For an atom A and any node N, if a leaf L in the tree is reachable through N.occurs[A] then L is an A-leaf in the subtree at N. Also, *if N is an A-node* then *all* A-leafs in the subtree at N are reachable through N.occurs[A].

Pp Invariant: For any atom A and any A-leaf L with parent P, if the subtree at P contains any other A-leaf then L.pp $\neq$ P.

The Tree invariants characterize trees that represent theories. The other invariants ensure that the corresponding fields in the nodes have the correct values. Note that nothing is said about the nodes in the occurs list being in the tree, or that all intermediate internal nodes in the tree are in the occurs list. The occurs-invariant for non-A nodes ensure that trying to propagate A at those nodes does not change the tree. Once established, these invariants continue to hold at all times, except for certain lines of the code — such violations are mentioned explicitly.

In the description of a procedure, PRE are the assertions that hold when the procedure is called, POST are the assertions that hold when the call is completed, and HOW is an informal description of the procedure. Only those assertions are mentioned that are relevant to the procedure. Moreover, since all the invariants are supposed to hold at all times, they are not explicitly mentioned in PRE and POST after being established for the first time. "Also" in POST indicates that all assertions in PRE (except those explicitly noted) continue to hold. The variables are subscripted by PRE or POST when it is not clear from the context whether the value of interest is that of before or after the procedure call.

Wherever possible, we indicate the type information with input and output parameters of various procedures. Some commonly used types (and the values they denote) are:

INT: integers;

NODE: nodes in a tree;

LABEL: $\odot, \wedge, \vee$, and facts;

ATOM: atoms (represented by integers);

BOOL: either true (also **t**) or false (also **f**);

LIT: literals (represented by integers);

Since atoms are represented by integers, negative literals are denoted by negative integers and complement of a literal is its negation. The type of a list of elements of type T is denoted by T LIST; for example, NODE LIST is the type of lists that contain elements of type NODE.

The following functions, constants, and variables are used in the description of the algorithms:

1. Constant **Nil** denotes an empty (but initialized) structure of any type;
2. Global variable **Root** denotes the root node of the tree;
3. Function **Read** returns the next integer from the input;
4. Function **abs(I)** returns the absolute value of the integer I. We will be coding literals by integers in such a way that if I denotes a literal then abs(I) is its atom;
5. Function **Leaf?(N : NODE)** returns true iff N is a leaf node which is still active.
6. Function **CreateNode(label : LABEL, childs : NODE LIST)** creates and returns a new node whose label and the childs fields are set to the two arguments respectively.
7. Procedure **Delete(N : NODE)** removes the node N, which is guaranteed to have zero children, from the tree by removing it from the childs list of its parent, and setting its parent to nil. It also makes it disappear from its ancestors' occurs lists. (As we shall see later in the complexity section, N is not explicitly removed from the occurs lists, but merely tagged as "invisible".)
8. Procedure **Deactivate(N : NODE)** is similar to Delete, except that N does not disappear from ancestors' occurs lists and is tagged as "dummy". (This will allow accessing leaf nodes for ancestors of N without having to explicitly change their occurs lists to eliminate N.)
9. Procedure **ChangeParent(C,N,P : NODE)** changes the parent of C from N to P. It does this by removing C from N.childs, pushing C to P.childs, setting C.parent to be P. At a conceptual level, the occurs lists are also appropriately modified. (As we shall see later in the complexity section, it is not actually required to modify the occurs lists — tagging the nodes appropriately is sufficient.)

The following operations are defined for lists:

Head(L): returns the first active element of List L if L is not empty, otherwise returns Nil;

Push(N,L): returns the list obtained by adding item N to the front of list L;

AddQ(N,L): returns the list obtained by adding item N to the end of list L;

Pop(N,L): returns Head(L), which is removed from list L;

Count(L): returns the number of items in the list L.

We also allow iteration over elements of a list using the "for" construct. The operational semantics of the construct "**For** N **in** Exp **where** Cond **do** S" is:

1. evaluate Exp, which returns a list, say M;
2. traverse M while creating a sublist L of items that staisfy the boolean predicate Cond;
3. execute statement S for each item N in L.

The "where Cond" part of the construct may be omitted, in which case L is the same as M. Note that the list L is computed before the iteration starts.

As is usual, the **no-op** statement does nothing, the **exit** statement terminates the innermost loop or iteration, and the **return** statement terminates the procedure call (functions return the value of the argument of return). We also abbreviate some combinations of statements — for example: the construct "i++ $< n$" returns true if $i < n$ and increments the value of i by 1; the boolean construct "(P := N.parent).label = X" sets P to N.parent and returns true iff N.parent.label = X (if N does not have a parent then P is undefined and "false" is returned).

We use "tree" to denote the entire tree (rooted at Root), and "subtree at N" to denote the subtree that is rooted at node N. The input theory is always denoted by Γ, and $term(N)$ denotes the term represented by the subtree at N.

3.3.1 Tuple Encoding of Theories:

We present a compact encoding of a theory based on a mapping of its atoms to the integers from 1 to k, where there are k distinct atoms in the theory. This encoding facilitates the reading of input to the algorithm AFP and creating the appropriate data structures. The complexity of AFP is measured in the size of this encoding. Similar, though not identical, encodings have been used by various others researchers (c.f. [DG84, MSL92]). A theory is encoded using a sequence of integers, by mapping each atom to a distinct integer:

Definition 3.4 [Tuple Encoding] For any theory Γ and any bijection

$$g : \text{atoms}(\Gamma) \to \{2, \ldots, |\text{atoms}(\Gamma)| + 1\}$$

the tuple encoding $\langle\langle s \rangle\rangle$ of any subterm s of Γ is inductively defined as:

1. $\langle\langle \wedge \rangle\rangle = \langle 1 \rangle$; $\langle\langle \vee \rangle\rangle = \langle -1 \rangle$; $\langle\langle \odot \rangle\rangle = \langle\rangle$ (empty tuple);
2. $\langle\langle P \rangle\rangle = \langle g(P) \rangle$ and $\langle\langle \neg P \rangle\rangle = \langle -g(P) \rangle$ for any atom P;
3. $\langle\langle c(s_1, \ldots, s_n) \rangle\rangle = \langle\langle c \rangle\rangle \diamond \langle n \rangle \diamond \langle\langle s_1 \rangle\rangle \diamond \ldots \diamond \langle\langle s_n \rangle\rangle$, where c is a connective (either $\wedge$, $\vee$, or $\odot$), and s's are formulas.

■

Since $\mathbf{f}$ and $\mathbf{t}$ abbreviates $\vee()$ and $\wedge()$ respectively, it follows that $\langle\langle \mathbf{f} \rangle\rangle = \langle -1, 0 \rangle$ and $\langle\langle \mathbf{t} \rangle\rangle = \langle 1, 0 \rangle$.

For example, if the bijection g is given by

$$g(P) = 2 \text{ and } g(Q) = 3$$

then the theory

$$\Gamma = [\![\vee(\vee(Q, \mathbf{f}), \wedge(P, \vee(\neg P, Q)))]\!]$$

is encoded as

$$\langle\langle \Gamma \rangle\rangle = \langle 1, -1, 2, -1, 2, 3, -1, 0, 1, 2, 2, -1, 2, -2, 3 \rangle$$

It is easy to verify that each theory and formula has a unique tuple encoding. Using the tuple encoding makes it almost trivial to determine whether an atom has already been seen earlier in the input and to allocate appropriate space for arrays that are indexed by atoms. These operations are more expensive if the theory is used directly without any such encoding. Tuple encoding also provides a uniform representation of terms, namely, using sequences of integers. The next proposition shows that tuple encoding is also compact.

Proposition 3.5 *For any subterm s of any theory Γ, the size of $\langle\langle s \rangle\rangle$ is at most the size of s.*

Proof: [by induction on the construction of Γ] In the base case, when s is a literal, $\langle\langle s \rangle\rangle$ consists of a single integer. In the inductive case, $\langle\langle s \rangle\rangle$ consists of the code of its connective and the count and codes of its immediate subterms — in terms of size, the pair of parentheses delimiting these subterms is replaced by the count. Thus, there is no increase in size. ■

3.4 AFPC: Simplification and Propagation Rules Only

We present an algorithm AFPC (Algorithm for FPC) that reduces any given term to its irreducible form with respect to the rewrite system FPC. In other words, AFPC implements the simplification and propagation rules. We also argue the correctness and provide an upper bound on time complexity of AFPC.

As we remarked earlier, simplification rules (S rules) are easy to apply as soon as they become applicable, even while reading the input theory. However, this is not the case with propagation rules (P rules), since their instances interact with each other. To deal with this, we will maintain a list PQ of all nodes where a P rule may be applicable. To begin with, PQ contains all nodes that have leaf children. Afterwards, each node is removed from the list PQ and the corresponding P rule is applied, if it is still applicable. Again, the S rules are applied as soon as they become applicable. Since this may create more possibilities for applying P rules, the corresponding nodes are also added to the list PQ. The algorithm terminates when either the queue is empty or the theory reduces to $\{\mathbf{f}\}$ or $\{\ \}$.

AFPC calls ReadArgs for constructing the tree by reading the input theory, and InitPropagate for setting the Occurs and PQ lists, and parent and pp fields; it then keeps popping nodes from the PQ list and calling Propagate to apply a P rule, until the list becomes empty and the execution terminates. Recall that the algorithm also terminates as soon as the tree becomes an exception tree.

For correctness, we need an invariant that ensures that all potential sites for P rules are recorded in the list PQ. In order to establish the Occurs invariant, we will first establish an intermediate assertion, which does not hold after the occurs invariant is established:

PQ: list containing nodes where P rules are applicable (new nodes are always added at the end);

PQ Invariant: For any atom A and any A-leaf L with parent P, if L.pp $\neq$ P then P is in the list PQ.

Occurs intermediate-assertion: For any atom A, any A-leaf L, and any proper ancestor P of L, P.occurs[A] = Nil.

Note that PQ Invariant, together with PP invariant, ensure that once PQ list is empty, there are no more applicable propagations.

Procedure AFPC

```
/* reads the input theory and rewrites it to an irreducible form
      with respect to FPC
PRE: input I = ⟪Γ⟫
POST: Γ ⇒!_FPC term(Root)
      all invariants
HOW: read the theory, set data structures while recording all potential
      sites of P rules in the list PQ, and then apply the P rules
      until PQ becomes empty.  S rules are applied on the fly.  */
 1. Root := ReadArgs(⊙);
 2. InitPropagate;
 3. while (PQ ≠ Nil) do {
 4.   Propagate(Head(PQ));
 5.   Pop(PQ)
 6. }
```

end (AFPC).

Note that the node where P rules are applied (in line 4) is removed from list PQ (in line 5), since any new node added to PQ goes at the end. The node is not removed before propagation because of the PQ invariant. Also, nodes that are deactivated or deleted are ignored while iterating through the PQ queue (the same will hold in queues for the other rules, as well).

For a connective c, ReadArgs(c) reads the arguments of c and constructs the tree that represents the entire term. It reads each argument, which is a formula, by calling ReadFml. The subtrees representing these formulas are collected in a list, except that S rules are used to eliminate the formulas $\mathbf{t}$ and $\mathbf{f}$. In general, the tree returned by ReadArgs is obtained by adding the root labeled by c to this forest of subtrees. However, two cases are treated differently:

1. if the list contains only one subtree then that subtree is returned — this is justified by Rule $S3$;

2. either **t** of **f** may be obtained from using the S rules, even without reading all the arguments.

In the latter case, the rest of the arguments must be read and discarded.

Function ReadArgs (lab : LABEL) : NODE

```
/* reads the arguments for the connective lab and reduces using S rules
PRE: lab ∈ {∧, ∨, ⊙};
      input has prefix I = ⟨n⟩ ◇ ⟪ψ₁⟫ ◇ ... ◇ ⟪ψₙ⟫.
POST: input I has been read;
      the tree returned represents the logical term φ, such that:
          φ is irreducible with respect to the S rules, and
          ψ ⇒*_R φ using the S rules, where ⟪ψ⟫ = ⟪lab⟫ ◇ I;
HOW: reads each formula ψᵢ by calling function ReadFml;
      discards t and f, and collects the rest in list childs;
      applies S rules on the fly;
      returns node P whose subtree is the simplified formula.  */
n := Read;
i := 1; /* next input is ⟪ψᵢ⟫ */
childs := Nil; /* no child yet */
P := Nil; /* cannot return a value yet */
while (i++ ≤ n) and (P = Nil) do /* read and process ψᵢ */
  case (M := ReadFml) of {
      Tnode :  if lab = ∨ then P := Tnode /* rules S1∨, S3 */
               else No-op /* rule S2∧ */
      Fnode :  if lab = ∧ then P := Fnode /* rules S1∧, S3 */
               elseif lab = ⊙ then P := Ftree /* rules S1∧ */
               else No-op /* rule S2∨ */
      else :  Push(M,childs) /* new child */
  }
if (P ≠ Nil) then /* read and discard rest of the formulas */
  while (i++ ≤ n) do ReadFml
elseif (childs = Nil) and (lab ≠ ⊙)
  thenif lab = ∨ then P := Fnode /* ∨(∅) = f */
          else P := Tnode /* ∧(∅) = t */
elseif (Count(childs) = 1) and (lab ≠ ⊙)
  then P := Head(childs) /* rule S3 */
else P := CreateNode(lab,childs); /* no S rules apply */
return P;
```

end (ReadArgs).

ReadFml reads a formula and constructs the tree that represents it. If the formula is a literal, then the tree has a single node labeled by that literal; otherwise it reads the connective, calls ReadArgs to read its arguments, and returns the same tree returned by ReadArgs.

Function ReadFml : NODE

```
/* inputs a formula and simplifies using S rules
PRE: input has prefix I = ⟪ψ⟫.
POST: input I has been read;
      the tree returned represents the formula φ, such that:
          φ is irreducible with respect to the S rules, and
          ψ ⇒*_R φ using the S rules.  */
case (x := Read) of {
   1  :  return ReadArgs(∧) /* ⟪∧⟫ = 1 */
  -1  :  return ReadArgs(∨) /* ⟪∨⟫ = -1 */
```

```
    else :  return CreateNode(x,Nil) /* x is a literal */
}
```
end (ReadFml).

Given a tree representing a theory, InitPropagate sets the PQ list and the parent, pp and occurs fields so that the corresponding invariants are established. It makes two passes over the tree: the first pass (InitOccurs) is used to initialize all the relevant fields of Occurs arrays to Nil; only these initialized fields are accessed in the second pass (SetOccurs). As we shall see later, not requiring the initialization of all the occurs arrays will be important in our analysis of the time complexity of AFPC. InitPropagate does not change $term(Root)$.

Procedure InitPropagate
```
/* initializes the data structure needed for propagation
PRE: Tree invariants
POST: also, parent, occurs, pp, and PQ invariants
HOW: two passes, from leaves to the root, are used to establish
      the occurs invariant; first pass establishes the occurs
      intermediate-assertion, ensuring that only the initialized
      occurs fields are accessed in the second pass.  */
InitOccurs(Root); /* first pass */
PQ := Nil;
SetOccurs(Root); /* second pass */
```
end (InitPropagate).

InitOccurs recursively traverses the entire tree. For each A-leaf, it traverses the branch from the leaf to the root setting the occurs[A] fields of the nodes to Nil. It also sets the parent fields of each non-root node in the tree.

Procedure InitOccurs (N : NODE)
```
/* recursive first pass before propagation starts
PRE: N is not a leaf;
      subtree at N satisfies tree invariants
POST: subtree also satisfies parent invariant and
      occurs intermediate-assertion.  */
for each M in N.childs do {
  M.parent := N; /* parent invariant established */
  if Leaf?(M) then { /* occurs intermediate-assertion for M */
      R := M;
      A := abs(M.label); /* M is an A-leaf */
      while (R ≠ Root) do { /* walk up to the root */
          R := R.parent;
          R.occurs[A] := Nil;
      }
  } else InitOccurs(M);
}
```
end (InitOccurs).

SetOccurs also recursively traverses the entire tree in preorder. For each A-leaf, it traverses the branch from the leaf to the root ensuring that each non-root node is in the occurs[A] list of its parent (if not, the node is added to the front of the list). It also sets the pp fields of each leaf to itself, and constructs the list PQ containing all those nodes in the tree which have leaf children.

Procedure SetOccurs (N : NODE)

```
/* recursive second pass before propagation starts
 PRE: N is not a leaf;
      subtree at N satisfies tree invariants,
          and occurs intermediate-assertion
 POST: subtree also satisfies occurs (but not occurs intermediate),
          pp, and PQ invariants */
  1. for each M in N.childs do {
  2.   if Leaf?(M) then {
  3.       M.pp := M; /* pp invariant established,
               since M has not been propagated */
  4.       R := M;
  5.       A := abs(M.label); /* M is an A-leaf */
  6.       while (R ≠ Root) do { /* P & R pair walks up to the root */
  7.           P := R.parent;
  8.           if (R = Head(P.occurs[A]))
  9.               then exit /* R is already in the occurs list of P */
 10.           else Push(R, P.occurs[A]);
 11.           R := P;
 12.       }
 13.   } else SetOccurs(M);
 14. }
 15. if N.leafs ≠ Nil then Push(N, PQ) /* PQ invariant */
```
end (SetOccurs).

Since the tree is traversed in preorder and Push adds elements at the front of a list, only the head node needs to be checked in line 8. Also, if the node is found in the list then there is no need to continue traversing the branch since the remaining nodes already satisfy the required condition. Note that N.leafs is set in ReadArgs, while reading the input theory.

For any node N, Propagate(N) propagates each leaf child M of N whose pp field is not set to N. After propagation, the pp field of M is set to N, as there is no other node in the subtree at N having the same atom in the label as that of M.

Procedure Propagate (N : NODE)
```
 /* propagate each leaf child of N and reduce using S rules
 PRE: term(Root) is irreducible with respect to S rules
 POST: also, for each M in (N.leafs)_PRE ∩ (N.leafs)_POST:  M.pp = N;
       term(Root_PRE) ⇒*_R term(Root_POST) using S and P rules.
 HOW: propagate each leaf of N, applying S rules on the fly */
  1. if Leaf?(N) then return;
  2. for each M in N.leafs where M.pp ≠ N do {
          /* otherwise M has already been propagated;
          next decide whether to substitute by t or by f.  */
  3.   val := (N.label = ∨);
  4.   if (M.label > 0) then val := not val;
  5.   A := abs(M.label);
  6.   PropAtom(A,val,N,M); /* do the propagation */
  7.   if Leaf?(M) then
  8.       N.occurs[A] := Push(M,Nil); /* all other A-leaves are gone */
  9.       M.pp := N; /* M has been propagated in the subtree at N */
 10.   }
 11. }
```
end (Propagate).

The check in line 7 is required since M may be deactivated by PropAtom. Line 8 re-establishes the Occurs invariant which was possibly violated by PropAtom, which had set N.occurs[A] to nil. The juggling with val in lines 3 and 4 ensures that the replacements by truth constants are done correctly. $M.pp = N$ in the postcondition does not apply to the leaves added to N during the call to the procedure. However, if any new leaves are added then N is also again added to list PQ, and these leaves get propagated during the next call of Propagate(N). Since new leaves are always added to the front, the for loop in line 2 can be stopped when the first node M with $M.pp = N$ is found.

PropAtom(A,val,N,M) propagates the A-leaf M in the subtree at N. It recursively traverses all the Occurs[A] lists in the subtree replacing each A-leaf (which is not equal to M) by a truth constant determined by val and rewriting the tree using S rules. Since all A-leaves, except M, are deactivated by S rules, the Occurs[A] lists are reset to Nil. As we saw above, the correction for M is made in Propagate.

Procedure PropAtom (A : ATOM; val : BOOL; N,M : NODE)

```
/* substitute each occurrence of A in the subtree at N
      by val and reduce using S rules;
      don't change M, which is the source of propagation
PRE: N is not a leaf;
      term(Root) is irreducible with respect to S rules
POST: also, occurs invariant may not hold for leaf M - node N pair.
      term(Root_PRE) ⇒*_R term(Root_POST) using S and P rules;
      except for possibly M, there is no A-leaf in the subtree at N;
NOTE: N may be a deactivated node.
HOW: find occurrences of A using the occurs[A] lists recursively */
 1. for each R in N.occurs[A] where R ≠ M do
 2.   if (R.label = A) then Simplify(R,val) /* R is a A-leaf */
 3.   elseif (R.label = -A) then Simplify(R, not val)
          /* R is a A-leaf */
 4.   else PropAtom(A, val, R, M); /* R is not a leaf */
 5. N.occurs[A] := Nil; /* all leaves, except for M, are gone;
          correction for M is done in Propagate */
```

end (PropAtom).

Simplify(N,val) replaces the label of leaf N by the truth-constant val and then reduces the tree using S rules. It does so by traversing all nodes in the branch from N to Root, while replacing by truth constants those nodes where S rules apply, and deactivating all the nodes in their subtrees. Note that the first node that is not so replaced has one less child than before, creating the possibility of applying rule S3.

Procedure Simplify (N : NODE; val : BOOL)

```
/* change label of N to val (using P rule) and reduce using S rules
PRE: term(Root) is irreducible with respect to S rules;
      N is a leaf
POST: also, term(Root') ⇒*_R term(Root_POST) using S rules alone;
      where Root' is obtained from Root_PRE by changing the label of
          N to val.
HOW: find the highest node (R) that can be replaced by either t (rule
      S1_∨) or f(rule S1_∧); apply either S2_∨ or S2_∧ to remove node R from
      its parent (P); if non-root P has a single child then apply S3 */
 1. if val then lab := ∨ else lab := ∧;
      /* select appropriate S1 and S2 rules */
 2. R := N;
 3. while (R.parent.label = lab) do R := R.parent;
      /* R cannot be the root */
 4. P := R.parent; /* R will be removed from the tree */
 5. DeleteTree(R); /* use the selected rules and S3 */
 6. if P=Root and Count(P.childs) = 0 then
```

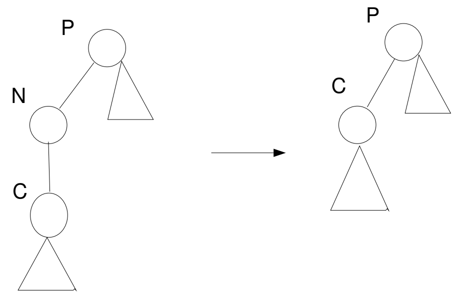

Figure 3.5: Collapse(N)

```
        /* rectify incorrect removal of R */
  7.   if val then Root := Ftree else Root := Ttree
  8. elseif (Count(P.childs) = 1 ) then Collapse(P) /* rule S3 */
```
end (Simplify).

In deactivating the nodes in line 5 of Simplify, the Ftree may be erroneously changed to Ttree — this is rectified in line 7. Also, $term(Root_{PRE}) \Rightarrow_R term(Root')$ using a single application of a P2 rule.

Collapse(N), called when node N has exactly one child, applies rule S3 by replacing the node by the child (see Figure 3.5). If this child is a leaf, then there is a possibility of using a P rule on its new parent.

Procedure Collapse (N : NODE)
```
 /* replace N by its only child
 PRE: N is not the root and N has a single child
 POST: node N is removed from the tree;
       term(Root_PRE) =>_R term(Root_POST) using a
       single application of S3 */
  1. C := Head(N.childs); /* C is the only child of N */
  2. P := N.parent;
  3. if Leaf?(C) then AddQ(P, PQ)
        /* for PQ invariant, since C.pp ≠ P */
  4. ChangeParent(C,N,P); /* make P the parent of C */
  5. Deactivate(N); /* remove N from the tree */
```
end (Collapse).

DeleteTree(N) recursively deletes all the nodes in the subtree at node N.

Procedure DeleteTree (N : NODE)
```
 /* remove the entire subtree at N
 PRE: N is not the Root
 POST: subtree at N is removed from the tree */
  1. if not Leaf?(N) then
  2.   for each C in N.childs do DeleteTree(C);
  3. Delete(N);
```
end (DeleteTree).

3.4.1 Correctness of AFPC

We now prove the correctness of AFPC by stating and proving the correctness claims for the various procedures that are called by AFPC. In general, these claims are more detailed forms of the corresponding PRE and POST assertions. We will first prove the correctness of ReadArgs, then of InitPropagate, and then of Propagate. We would assume that the precondition of AFPC holds, i.e., the input is a correct encoding of some theory. Recall that the algorithm terminates if the tree becomes an exception tree.

Since most procedures are (mutually) recursive, the proofs are usually based on induction using the recursion tree. A recursion tree represents the nesting of recursive calls in any particular execution of the algorithm — procedure call P is a child of procedure call Q iff P is called from Q. For proving a property for each node in the recursion tree, we use induction in two ways:

1. *Induction on the level of recursion:* the base case is for the root node;
2. *Induction on the depth of recursion:* the base case is for the leaf nodes.

Lemma 3.6 (Correctness of ReadFml and ReadArgs) *During the execution of ReadArgs($\odot$) on any input tuple that correctly encodes some theory:*

1. *Each call of ReadFml reads a non-empty prefix $I = \langle\!\langle \psi \rangle\!\rangle$ of input for some formula ψ, and returns a tree that represents a formula, say φ, such that:*
   (a) φ is irreducible with respect to the S rules, and
   (b) $\psi \Rightarrow_R^* \varphi$ using the S rules;
2. *For any $c \in \{\wedge, \vee, \odot\}$, each call of ReadArgs(c) reads a non-empty prefix $I = \langle n \rangle \diamond \langle\!\langle \psi_1 \rangle\!\rangle \diamond \ldots \diamond \langle\!\langle \psi_n \rangle\!\rangle$ of input for some n and some formulas $\psi_1, \ldots, \psi_n$, and returns a tree that represents a logical term, say φ, such that*
   (a) φ is a theory iff $c = \odot$,
   (b) φ is irreducible with respect to the S rules, and
   (c) $\psi \Rightarrow_R^* \varphi$ using the S rules, where $\langle\!\langle \psi \rangle\!\rangle = \langle\!\langle c \rangle\!\rangle \diamond I$.

Proof: Since the two functions are mutually-recursive, we will prove both claims by simultaneous induction on the depth of recursion in any particular execution of ReadArgs($\odot$). Before that, we make some observations:

1. The variable P in ReadArgs is assigned a value exactly once (after initialization in line 4), which is then returned by the function in line 22.
2. The only rewrite rules that we need to consider in this proof are the S rules.

(Base case for ReadArgs) Since there is no call to ReadFml, $n = 0$ and $\psi = \varphi = c(\emptyset)$.

(Base case for ReadFml) Since there is no call to ReadArgs, $x = \langle\!\langle \alpha \rangle\!\rangle$ and $\psi = \varphi = \alpha$ for some literal α.

(Inductive Case for ReadFml) Since there is exactly one call to ReadArgs, whose value is returned back, the claim follows directly from the inductive assumption for ReadArgs.

(Inductive Case for ReadArgs) Since there is at least one call to ReadFml, $n > 0$. The while loops of lines 5 and 15 are executed exactly n times in total, where each iteration calls ReadFml once. By the inductive assumption for ReadFml, $\langle\!\langle \psi_i \rangle\!\rangle$ is read in the ith iteration. This proves the first claim; we now prove the rest.

By the inductive assumption for ReadFml, the call to ReadFml that reads $\langle\!\langle \psi_i \rangle\!\rangle$ returns the tree representing the formula φ_i such that $\psi_i \Rightarrow_R^* \varphi_i$ and φ_i is irreducible with respect to S rules. Thus, $\psi = c(\psi_1, \ldots, \psi_n) \Rightarrow_R^* c(\varphi_1, \ldots, \varphi_n)$. All we need to show is that $c(\varphi_1, \ldots, \varphi_n) \Rightarrow_R^* \varphi$ and that φ is irreducible (both with respect to S rules). We do a case analysis on all the possibilities:

$c = \odot$ and $\varphi_i = \mathbf{f}$ for some i: $c(\varphi_1, \ldots, \varphi_n) \Rightarrow^*_R \odot(\mathbf{f})$ (irreducible) using Rule $S1_\odot$. P gets the correct value in line 10 of the algorithm.

$c = \wedge$ and $\varphi_i = \mathbf{f}$ for some i: $c(\varphi_1, \ldots, \varphi_n) \Rightarrow^*_R \mathbf{f}$ (irreducible) using Rules $S1_\wedge$ and $S3$. P gets the correct value in line 9.

$c = \vee$ and $\varphi_i = \mathbf{t}$ for some i: $c(\varphi_1, \ldots, \varphi_n) \Rightarrow^*_R \mathbf{t}$ (irreducible) using Rules $S1_\vee$ and $S3$. P gets the correct value in line 7.

Otherwise, let $B = [\![\varphi_i \mid i \in 1, \ldots, n; \varphi_i \notin \{\mathbf{t}, \mathbf{f}\}]\!]$. It follows from the inductive assumption that the childs list contains exactly the trees that represent the formulas in B. Moreover, $c(\varphi_1, \ldots, \varphi_n) \Rightarrow^*_R c(B)$ using Rules $S2_\vee$ and $S2_\wedge$. The various possibilities are:

$c = \wedge$ and $B = \emptyset$: $c(B) = \mathbf{t}$ is irreducible. P gets the correct value in line 18.

$c = \vee$ and $B = \emptyset$: $c(B) = \mathbf{f}$ is irreducible. P gets the correct value in line 17.

$c \neq \odot$ and $B = \{\varphi_i\}$ for some i: $c(B) \Rightarrow^*_R \varphi_i$ (irreducible by the inductive assumption) using Rule $S3$. P gets the correct value in line 20.

otherwise: $c(B)$ is irreducible as no rule applies (note that each formula in B is irreducible, using the inductive assumption). P gets the correct value in line 21.

Thus, in all cases, $\psi \Rightarrow^*_R \varphi$ and φ is irreducible, both with respect to S rules. ∎

It follows directly from Lemma 3.6 that the function ReadArgs($\odot$) on input $\langle\!\langle \Gamma \rangle\!\rangle$ (where Γ is any theory) returns a tree representing a theory, say Δ, such that

1. the tree satisfies Tree invariants;
2. Δ is irreducible with respect to S rules;
3. $\Gamma \Rightarrow^*_R \Delta$ using S rules.

The invariants established by ReadArgs($\odot$) are precisely the preconditions for InitPropagate and InitOccurs.

Proposition 3.7 (Correctness of InitOccurs) *For each call to InitOccurs(N) during the execution of InitPropagate, N is non-leaf node, and after the call is completed:*

1. *for any parent-child pair P and C in the subtree at N, C.parent = P;*
2. *for any atom A, any A-leaf L in the subtree at N, and any proper ancestor P of L in the tree, P.occurs[A] = Nil.*

Proof: The first claim is proved by induction on the level of recursion. In the base case, when N = Root, it is not a leaf since the tree is not exception. In the inductive case, any recursive call to InitOccurs is made only for non-leaves.

The other claims are proved by induction on the depth of recursion. In the base case, since there is no recursive call to InitOccurs, each node in N.childs is a leaf. Claim 1 follows from line 2 of the procedure, while claim 2 follows from the while loop of line 6. The same argument proves the inductive case as well.

Note that both the inductions above terminate, since the tree at node N is finite. ∎

It follows directly from Proposition 3.7 that the parent invariant and occurs intermediate-assertion hold after the call to InitOccurs(Root) is completed.

Proposition 3.8 (Correctness of SetOccurs) *For each call to SetOccurs(N) during the execution of InitPropagate, N is a non-leaf node, and after the call:*

1. *for any parent-child pair P and C in the tree and any atom A, if C is in P.occurs[A] then there is a A-leaf in the subtree at C;*
2. *for any atom A, if N has a A-leaf child L then C is in P.occurs[A] for any parent-child pair P and C, both of which are ancestors of L — if C is also an ancestor of N then C = Head(P.occurs[A]);*
3. *for any leaf L with parent P, both of which are in the subtree at N, L.pp $\neq$ P and P is in the list PQ.*

Proof: The first claim is proved as in the previous proposition. The other claims are easily proved by induction on the depth of recursion.

1. line 10 is the only place where Occurs lists are modified — the A-leaf in the subtree at C (R in the procedure) is M.
2. follows from the while loop in line 6. C is at the head, when it is also an ancestor of N, since the tree is being traversed in preorder and Push(N,L) adds item N to the head of the list L. This does not hold in general, since the subtree at N may have multiple A-leaves.
3. follows from lines 3 and 15, respectively.

The exit from the loop in line 9 is justified by claims 1 and 2 above. ∎

It follows from Proposition 3.8 that the Occurs, Pp, and PQ invariants hold after the call to SetOccurs(Root) is completed. The correctness of InitPropagate follows directly from Propositions 3.7 and 3.8.

Lemma 3.9 (Correctness of Collapse) *For any call of Collapse(N):*

1. *N is not the root and has only one child;*
2. $term(Root_{PRE}) \Rightarrow_R term(Root_{POST})$ *using a single application of S3 rule to the subtree at N;*
3. *no invariants are violated.*

Proof: Collapse is called only in line 8 of Simplify, which ensures claim 1. Claim 2 is ensured by lines 4 and 5 of the procedure. The only possible violation of an invariant, namely the PQ invariant, is pre-empted in line 3. ∎

Lemma 3.10 (Correctness of Simplify) *For any call of Simplify(N,val)*

1. *N is a leaf;*
2. *term(Root) is irreducible with respect to S rules (before and after the call);*
3. $term(Root_{PRE}) \Rightarrow_R term(Root')$ *using a P rule and* $term(Root') \Rightarrow_R^* term(Root_{POST})$ *using S rules, where Root' is obtained from* $Root_{PRE}$ *by changing the label of N to val.*

Proof:

1. Simplify is called only in lines 2 and 3 of PropAtom — M is a leaf in both cases.
2. By induction on the number of calls of Simplify made before this call. For the base case, $term(Root)$ is irreducible with respect to S rules, since it is the output of ReadArgs($\odot$). For the inductive case, the tree is irreducible with respect to S1 and S2 rules, since Simplify does not label any node by a truth-constant. If P is not the root, then it must have at least two children before the call (otherwise the tree was not irreducible with respect to S3). Thus, it has at least one child after line 3. If it has exactly one child then S3 is applied in line 7.

3. Line 5 of the algorithm ensures that $term(Root_{PRE}) \Rightarrow_R term(Root')$ using a P rule and $term(Root') \Rightarrow^*_R term(Root_{POST})$ using S rules, except in the following case: when applying S rules (line 5) leaves only the child **t** or **f** of the root. If this child is **t** then it is correctly removed using rule $S2_\wedge$; otherwise, its removal is incorrect — this error is rectified in line 7.

■

Lemma 3.11 (Correctness of Propagate) *For any call of Propagate(N)*

1. *term(Root) is irreducible with respect to S rules (before and after the call);*
2. *$term(Root_{PRE}) \Rightarrow^*_R term(Root_{POST})$ using S and P rules;*
3. *for each leaf M of N (which is a leaf both before and after the call), M.pp = N.*

Proof: For any leaf M of N (both before and after), Propagate calls PropAtom which traverses the occurs lists corresponding to A (which is the atom in the label of M) and calls Simplify on all leaves that it encounters. It also removes these occurs lists after the traversal. The correctness of PropAtom follows from the correctness of Simplify and the occurs invariant — the occurs invariant ensures that all the A-leaves in the subtree at N are accessed. The correctness of Propagate follows directly from this. Note that M.pp is set to N in line 9 of the procedure. ■

We are now ready to prove the correctness of AFPC:

Theorem 3.12 (Correctness of AFPC) *For any theory Γ, the procedure AFPC on input $\langle\langle\Gamma\rangle\rangle$ terminates with a tree representing a theory Δ such that $\Gamma \Rightarrow^!_{FPC} \Delta$.*

Proof: Lemmas 3.6 and 3.11 and Propositions 3.8 and 3.7 show the correctness of ReadArgs, Propagate, and InitPropagate (procedures called by AFPC), respectively. Thus, $\Gamma \Rightarrow^*_R \Delta$ using S and P rules, and Δ is irreducible with respect to S rules. There are two cases:

1. Δ is represented by an exception tree: it is trivially irreducible with respect to P rules;
2. otherwise: list PQ must be empty. It follows from PQ and pp invariants that Δ is irreducible with respect to P rules.

In both cases, Δ is irreducible with respect to P rules. Thus, $\Gamma \Rightarrow^!_{FPC} \Delta$. ■

3.4.2 Complexity of AFPC

We show that the time complexity of AFPC is quadratic in the size of the input theory. More precisely, for an input theory Γ, AFPC runs in time $O(nk)$, where n is the size of tuple $\langle\langle\Gamma\rangle\rangle$ and k is the depth of Γ, which is defined to be the depth of the tree representing Γ. Intuitively, the amount of time spent on each node is at most the level of the node in the tree.

The basic idea in proving the complexity result is as follows. We will first give details of the various data structures and show how various operations on them can be performed efficiently. In particular, we will define two ways of marking deleted nodes, which are not immediately removed from the Occurs lists. We then show that although the input theory is read in linear time, initializing the occurs arrays costs $O(nk)$, since the entire branch from the node to the root is traversed for each node in the tree. We then bound the total number of nodes that are added to PQ, and the total cost of all calls to Simplify and PropAtom, which gives us the bound on the running time of AFPC. Intuitively, the most amount of work spent on any node is dominated by the number of times it has to be raised in the tree so as to become the root.

Definition 3.5 The depth $\underline{\text{depth}(t)}$ of any logical term t is defined inductively as follows:

- if t is a literal then depth$(t) = 1$;
- for any bag B of formulas and any connective c, depth$(c(B)) = 1 + max\{0, \text{depth}(\psi) \mid \psi \in B\}$.

■

Note that depth k is at most the size n. We also use the parameter m to denote the number of distinct atoms $|\text{atoms}(\Gamma)|$ in the theory Γ. We use "initial tree" for referring to the tree returned by ReadArgs($\odot$).

For any data structure, we assume that memory for it is allocated in the procedure where it is first used. We do not require that memory be initialized when allocated. Such allocation of memory is a constant-time operation, independent of the size of the memory being allocated. The disadvantage is that any location in the allocated memory must be explicitly initialized before its use in the algorithm. In particular, while O(m) space for the array N.occurs for any node N can be allocated in constant time, any particular array entry must be initialized before its use. This initialization is explicitly done in InitOccurs.

The list PQ is a circular list so that both the ends can be accessed in constant time. The children of a node are kept in the subs and leafs lists, both of which are doubly-linked lists that explicitly maintain count of the number of items in them. The childs list is not explicitly maintained, but is implicitly viewed as a union of these two lists. Thus, deleting a node from the tree and retrieving the count of leafs, subs, and childs are each constant-time operations. Leaves and non-leaves are kept separate even while creating the list of children (childs) in ReadArgs.

The occurs lists are singly-linked lists with no counts. The only operations on them are traversing (in PropAtom), making them Nil (in PropAtom and InitOccurs), and adding an item at the front (in Propagate and SetOccurs). Nodes that are deactivated or deleted from the tree are not immediately removed from the occurs lists. Such nodes are marked, as described below, to distinguish them from the (undeleted) nodes in the tree:

Invisible: A node is marked invisible when it is deleted from the tree in a call to DeleteTree.

Dummy: A node is marked dummy when it is deleted from the tree in a call to Deactivate.

At any step of the algorithm, since all descendents (in the initial tree) of a invisible node are either invisible or dummy, invisible nodes can be completely ignored while traversing all lists. However, a dummy node cannot be ignored in an occurs list (though it can be skipped in other queues), unless it is an original leaf, since some of its descendents (in the initial tree) may still be in the tree, and we do not reset the occurs arrays to by-pass these no longer active nodes. Nodes that are to be ignored in a list because of being marked Invisible or Dummy are actually removed as encountered while traversing that list; this saves us the cost of *searching* for them.

ChangeParent(C,N,P) is called at only one place — line 4 of Collapse. Since N is deactivated (and marked as "dummy") in the next line, ChangeParent can leave the node N in the occurs list of P so any occurrences of leaf descendants of P can be accessed through its occurs list, just as before. Thus, there is no need to modify occurs lists during this call to ChangeParent.

Thus, each of the following operations is performed in constant-time using the above data structures:

- Read, Abs
- Count (for childs, leafs, and subs), Head, Push, Pop, AddQ
- Leaf?, CreateNode, Deactivate, Delete
- ChangeParent

Lemma 3.13 (ReadArgs) *Any call to ReadArgs and ReadFml that reads input tuple I:*

1. *takes $O(p)$ time, and*

2. *returns a tree of depth at most k containing at most p nodes;*

where $p = |I|$ and k is the depth of the term encoded by I.

Proof: Ignoring the while loops in ReadArgs and the recursive calls to ReadArgs in ReadFml, each of the two functions takes constant time, consumes exactly one integer from the input, and creates at most one new node. Each iteration of the while loops generates exactly one recursive call to ReadFml; if the cost of this call is ignored then each such iteration takes constant time. Moreover, each nesting of a pair of mutually recursive calls of the two functions increases the depth of the tree by at most 1. The lemma then follows from a straightforward induction on the depth of the mutual recursion in any particular execution. ∎

Lemma 3.14 (InitPropagate) *The call to InitPropagate*

1. *takes O(nk) time, and*
2. *creates occurs lists such that the total number of items in all of them is at most nk.*

Proof: It follows from Lemma 3.13 that the input tree has at most n nodes and at most k depth. InitPropagate makes two passes over the entire tree, each pass visits a node exactly once. The most costly visit is for a leaf, for which the entire branch to the root may be traversed, taking O(k) time. Thus, the total time is O(nk). The second claims follows directly from this bound. ∎

Now that the tree has been constructed and data structures initialized within the desired time limits, we make some observations on the execution of the algorithm:

Observations:

1. Once created by ReadArgs($\odot$), the size and the depth of the tree never increases. This holds because no new node is ever created later.
2. The parent of any node at any time in the tree must be among its ancestors at all previous times (after creation by ReadArgs($\odot$)). This holds because the only way to change parents is by collapsing a node in Collapse.
3. The procedure Collapse can be called at most n times, and each call takes constant time. This holds because each call deletes a node from the tree, and there are at most n nodes to begin with. Note that each of the operations Head, AddQ, Leaf?, ChangeParent, and Deactivate takes constant time.
4. The total number of items to be ever added in list PQ is at most 2n. Initially, since any non-leaf node can be in PQ, there can be at most n (a very liberal count!) of those. Later, the only addition of a new item to PQ is in Collapse which, from the previous observation, can happen at most n times.
5. For any node N, the set of its leaf children M such that M.pp $\neq$ N forms an initial segment of the list N.leafs. This can be proved by induction on the number of call that have been made to Propagate(N) so far. In the base case, no call was made, and M.pp = M for each leaf children of N. In the inductive case, the last call to Propagate(N) ensured that M.pp = N for all leaf children at that time. Any new child leaf added after that (by Collapse) is added at the front of the list Leafs and has M.pp $\neq$ N. Thus, the claim follows.

Lemma 3.15 *The procedure PropAtom is called at most O(nk) times from outside PropAtom.*

Proof: PropAtom is initially invoked in Propagate on node M iff M.pp $\neq$ N, the parent of M. After this call is completed, M.pp is set to N and is never changed until M gets a different parent (and PropAtom is called again). It follows from observation 2 that PropAtom on M can be thus called at most once for each of its ancestor in the initial tree. Since there are at most n nodes and at most k ancestors of each of them in the initial tree (which never grows, see observation 1), PropAtom is called at most O(nk) times from outside. ∎

Lemma 3.16 *The total time over all calls of Simplify is O(n).*

Proof: Each call to DeleteTree removes at least one node from the tree; if it removes p nodes then it takes O(p) time. Each node R traversed in the while loop in line 3 of Simplify is later removed by DeleteTree. Using observation 3, if a call to Simplify removes p nodes from the tree then it takes O(p) time. Since there are at most n nodes to begin with, the total time over all calls of Simplify is O(n). ∎

Observation 6. The total number of items ever added to all occurs lists is O(nk). From Lemma 3.14 , there were at most O(nk) items in the occurs lists to begin with. A new item is added only in each iteration of the while loop in Propagate, and nowhere else. It follows from Lemma 3.15 that the total number of such iterations is bounded by O(nk).

Lemma 3.17 *The total time taken by all calls to PropAtom is O(nk).*

Proof: If follows from Lemma 3.16 that the total time over all calls of Simplify is O(n), which is subsumed by O(nk). Thus, we can ignore these calls in computing the time taken by PropAtom. Apart from making these calls, PropAtom only traverses the occurs lists. Any item in the occurs list ever traversed by PropAtom is deleted and never traversed again. Thus, the total time taken by all calls to PropAtom is bounded by the total number of items ever added to occurs lists; which is O(nk) using Observation 6. Thus, the total time taken by all calls to PropAtom is O(nk). ∎

Theorem 3.18 (Time complexity of AFPC) *For any input theory Γ, algorithm AFPC takes O(nk) time, where n is the size of tuple $\langle\!\langle\Gamma\rangle\!\rangle$ and k is the depth of (the tree representing) Γ.*

Proof: From Lemma 3.13 and 3.14, ReadArgs($\odot$) and InitPropagate takes O(nk) time each. Since the node N is removed from list PQ when Propagate(N) is called, it follows from observation 4 that Propagate is called at most 2n times. It follows from observation 5 that the number of M's examined in the list N.leafs (during each call to Propagate) is at most 1 more than the number of times PropAtom is called; thus, the time for Propagate is dominated by the time for the calls to PropAtom. Thus, it follows from Lemma 3.17 that AFPC takes O(nk) time. ∎

3.5 AFPL: Adding Lifting Rules

We now extend algorithm AFPC for dealing with lifting rules. We also prove the correctness of the new algorithm AFPL and provide a quadratic bound on its running time.

Lifting rules (L rules) are implemented similarly to the P rules: the basic idea is to use lists for recording potential sites where L rules can be applied, and to continue rewriting until these lists and the list PQ are all empty or the tree becomes an exception tree. Each L rule is implemented using a new procedure, which is only called from the main algorithm. As before, S rules are applied as soon as they become applicable.

The two new global lists to keep track of nodes where the L rules may apply are:

L1Q: list of nodes where rule L1 may apply (Recall that the L1 rule $\wedge(\wedge(\alpha, B_1), B_2) \Rightarrow \wedge(\alpha, \wedge(B_1), B_2)$ is said to be applicable to the node at the root of the subtree $\wedge(\alpha, B_1)$.);

L2Q: list of nodes where rule L2 may apply.

We also need two new invariants to ensure that all potential sites for applying L rules are considered:

L1 invariant: if rule L1 applies to any node N then N is either in list PQ or in list L1Q.

L2 invariant: if rule L2 applies to any node N then N is either in list PQ, in list L1Q, or in list L2Q.

These invariants are first established by InitPropagate: As the input theory is read, it is represented as a tree, while recording each node with a leaf child in the list PQ. Since L1 is applicable to a node only if it has a leaf grandchild, this sets up the L1 invariant as well. Finally, if L2 is applicable to a node N, then N has to have a leaf child (since N must have at least two children because of tree invariant), and hence N is in the PQ list, establishing L2 invariant.

Thereafter, nodes are repeatedly removed from the lists PQ, L1Q, and L2Q (in that order) and the corresponding rules are applied if they are still applicable. As suggested by the L1 and L2 invariants, each node removed from PQ is added to list L1Q, and each node removed from L1Q is added to L2Q.

For moving leaves while applying rule L1, we need a variant, MoveLeaf, of ChangeParent that also modifies the occurs lists. Also, while applying rule L2, we need to merge the occurs lists of two nodes:

Procedure MoveLeaf(C,N,P : NODE) moves the leaf C from node N to node P. It does this by removing C from N.childs, pushing C to P.childs, setting C.parent to be P. `N.occurs[A]` is set to nil for the atom A of the literal at C (i.e., `A = abs(C.label)`)[1]. Also, if N is in `X.occurs[A]` at some node X for atom A then N there is replaced by C.

Procedure MergeOccurs(N,M : NODE) merges (at the conceptual level) the lists in the Occurs array of M to the corresponding lists in the Occurs array of N (as we show in the complexity section, the two arrays are not explicitly merged). The effect of merging is that for each atom A, if M is an A-node then all the A-leaves in the subtree at M also become accessible through N.Occurs[A].

In the complexity section, we will show how to efficiently modify occurs lists during MoveLeaf, and merge them in MergeOccurs.

Procedure AFPL

```
/* reads the theory and rewrites it to an irreducible form using
      S, P, and L rules
PRE: input I correctly encodes a theory, say Γ
POST: theory Δ represented by the tree is irreducible with respect to S,
      P, and L rules; Γ ⇒*_R Δ using S, P, and L rules;
      all invariants are satisfied */
Root := ReadArgs(⊙);
InitPropagate;
L1Q := L2Q : = Nil;
loop {
  if PQ ≠ Nil then { /* possible P rule */
      Propagate(Head(PQ));
      AddQ(Pop(PQ),L1Q);
  } elseif L1Q ≠ Nil then { /* possible L1 rule */
      Lift1(Head(L1Q));
      AddQ(Pop(L1Q),L2Q);
  } elseif L2Q ≠ Nil then { /* possible L2 rule */
      Lift2(Head(L2Q));
      Pop(L2Q);
  } else exit;
}
```

end (AFPL).

Lift1(N) first tests whether L1 rule is applicable to the node N; if yes, each leaf child of N is made a child of N's parent (see Figure 3.6). Since the list PQ was empty, the tree was irreducible with respect to P rules when Lift1 is called. This ensures that the occurs lists in N corresponding to its leaves are singletons; thus,

[1] This is safe because when MoveLeaf is invoked, C would already have been propagated at N (since PQ is empty) so there would only be at most one occurrence of A under N.

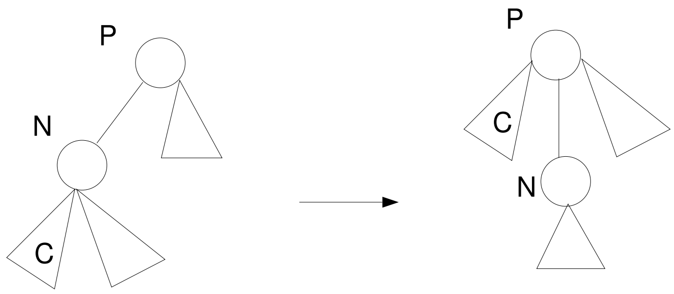

Figure 3.6: Lift1(N)

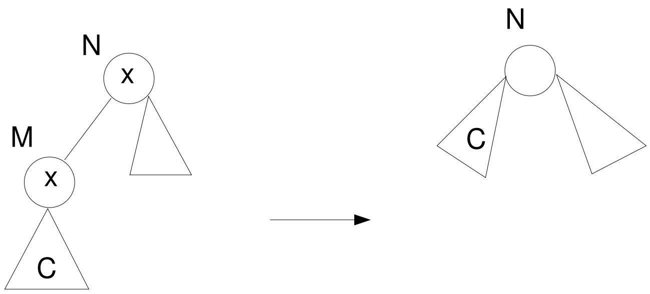

Figure 3.7: Lift2(N)

they are each set to Nil after the leaves are moved. If N is left with no child then it is removed from the tree using S1 and S2 rules; if it is left with only one child, then it can be collapsed using Rule S3. Since N's parent get new leaves, it becomes a potential site for using P rules.

Procedure Lift1 (N : NODE)

/* check and apply rule L1 and then reduce using S rules
PRE: $term(Root)$ is irreducible with respect to S rules;
      for each leaf child L of N: L.pp = N
POST: also, L1 rule does not apply at N
      $term(Root_{PRE}) \Rightarrow_R^* term(Root_{POST})$ using L1 and S rules */
if N $\neq$ Root and N.leafs $\neq$ Nil and [((P := N.parent).label =
   N.label or (P = Root and N.label = $\wedge$))] then {
   for each C in N.leafs do /* apply L1 rule */
       MoveLeaf(C,N,P);
           /* N has no leaf childs left */
   if N.childs = Nil then Deactivate(N)
   /* N is [[]]; apply either rule $S2_\wedge$ or $S2_\vee$ */
   elseif Count(N.childs) = 1 then Collapse(N); /* rule S3 */
   AddQ(P,PQ); /* P got at least one new leaf child */
}
end (Lift1).

Lift2(N) first tests whether L2 rule is applicable at node N; if yes, the only non-leaf child of N is removed from the tree by moving all its children to N (see Figure 3.7).

Procedure Lift2 (N : NODE)

```
 /* check and apply rule L2 and then simplify
 PRE: term(Root) is irreducible with respect to S rules
 POST: also, L2 rule does not apply at N
       term(Root_PRE) ⇒*_R term(Root_POST) using L2 and S rules */
  1. if Count(N.subs) = 1 and (M := Head(N.subs)).label = N.label then
       { /* apply L2 rule */
  2.   if M.leafs ≠ Nil then AddQ(N,PQ);
               /* N will get M's leaf children */
  3.   MergeOccurs(N,M);
  4.   for each C in M.childs do ChangeParent(C,M,N);
  5.   Deactivate(M);
  6. }
```

end (Lift2).

The second test in line 1 ensures that N is not the root. It is possible that node N, which was not an A-node for some atom A before the call to Lift2, becomes an A-node after the call. In that case, the occurs invariant may be violated for N and A after Lift2. MergeOccurs(N,M) prevents this by setting N.Occurs[A] correctly. For now (until we add the factoring rule in the next section), it is guaranteed that both N.Occurs[A] and M.Occurs[A] have been initialized since the A-leaf was a descendent of both N and M in the original tree. Since Lift2 is called only after no P rule applies, it follows that (just before the call):

1. none of the leaf children of N is an A-leaf;
2. M is an A-node (if it is not, then the Occurs invariant would not be violated, so the operations performed here can be ignored, even though they are performed);
3. N.Occurs[A] is either M or Nil (since N has only M as a non-leaf child.)

If N.Occurs[A] is M then all the A-leaves in the subtree at N are accessible through N.Occurs[A] even before the call (and continue to be so after the call) because of the Occurs invariant for M and A. Otherwise, these leaves become accessible after the call to MergeOccurs(N,M). Thus, the occurs invariant is satisfied in both cases.

There are two more cases when the L2 rule may become applicable. These occur at the end of procedures Simplify and Collapse, when P is left with only one non-leaf child which has the same label as P. Thus, we need to add the following lines at the end of Simplify:

```
elseif count(P.subs) = 1 and Head(P.subs).label = P.label then
       AddQ(P,L2Q);
```

and the following lines at the end of Collapse:

```
if count(P.subs) = 1 and Head(P.subs).label = P.label then
       AddQ(P,L2Q);
```

3.5.1 Correctness of AFPL

Our basic approach in proving the correctness remains the same as that for the algorithm AFPC. We need to first prove the correctness of Lift1 and Lift2, before proving the correctness of AFPL.

Lemma 3.19 (Correctness of Lift1) *For any call of Lift1(N):*

1. *term(Root) is irreducible with respect to S rules (before and after the call);*
2. *L.pp = N for each leaf child L of N;*
3. $term(Root_{PRE}) \Rightarrow_R^* term(Root_{POST})$ *using L1 and S rules;*
4. *L1 rule does not apply at node N after the call.*

Proof: If the test in lines 1 and 2 fails then Lift1 terminates without changing anything, since L1 rule does not apply at node N. Otherwise:

1. Since outputs of both Propagate and ReadArgs are irreducible with respect to S rules, term(Root) is irreducible with respect to S rules before the call. The only way to obtain either **t** or **f** is by lifting up all the children of N (i.e., all of them are leaves); in that case line 5 removes the truth constant by using rules S1 and S2. Thus, the output is irreducible with respect to S1 and S2 rules. The only potential for rule S3 is tried in line 6; thus the output is also irreducible with respect to to S3.
2. Directly from invariant PQ since the list PQ is empty whenever Lift1 is called;
3. All the changes made to the tree are by applying rules L1, S1, S2, and S3.
4. Since all the leaf children of N are lifted to its parent, rule L1 does not apply to node N (which may have been even deactivated) after the call.

This proves the lemma. ■

Theorem 3.20 (Correctness of AFPL) *For any theory* Γ*, the procedure AFPL on input* $\langle\langle\Gamma\rangle\rangle$ *terminates with a tree representing a theory* Δ *such that* Δ *is irreducible with respect to S, P, and L rules and* $\Gamma \Rightarrow_R^* \Delta$ *using S, P, and L rules.*

Proof: Any instance of rule L2 for which n = 0, i.e., there is no leaf child, is also an instance of rule S3. Since we are always keeping the tree irreducible with respect to S rules, rules L1 and L2 apply to a node only if it has at least one leaf child. Thus, both L1 and L2 invariants are established by InitPropagate since it adds all nodes with leaf children to the list PQ.

Lemmas 3.6, 3.11, 3.19 and Proposition 3.8 show the correctness of ReadArgs, Propagate, Lift1, and InitPropagate (procedures called by AFPL), respectively. The correctness of Lift2 follows directly from its code. Thus, $\Gamma \Rightarrow_R^* \Delta$ using S, P, and L rules, and Δ is irreducible with respect to S rules. There are two cases:

1. Δ is represented by an exception tree: it is trivially irreducible with respect to P and L rules;
2. otherwise: lists PQ, L1Q, and L2Q must be empty. It follows from PQ and pp invariants that Δ is irreducible with respect to P rules, and from L1Q and L2Q invariants that Δ is irreducible with respect to L1 and L2 rules, respectively.

In both cases, Δ is irreducible with respect to L and P rules. This proves the theorem. ■

3.5.2 Complexity of AFPL

We show that the complexity of AFPL is the same as that of AFPC, i.e., $O(nk)$, where n is the size of the input theory and k is its depth. We do this by looking at the extra work done in AFPL, as compared to AFPC, and showing that it is still bounded by $O(nk)$. The most difficult part is showing that the total cost of all calls to Lift2 has the correct bound, since each call requires merging two Occurs arrays. We resolve

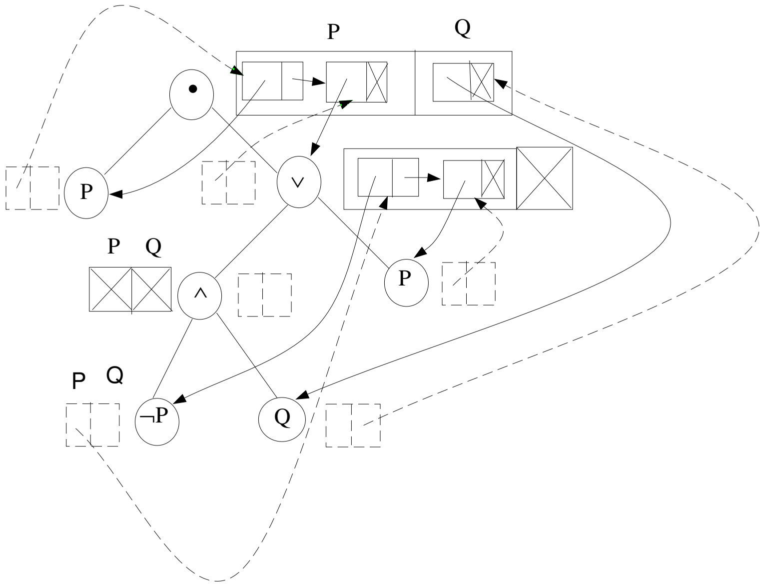

Figure 3.8: Back pointers for occurs lists

this by introducing back pointers for entries in the Occurs arrays, which are used for traversing the leaves of a subtree and correctly linking the corresponding Occurs entries in the two arrays to be merged. We also strengthen the Occurs invariant so that these back pointers are compactly represented.

We use the same data structures as in AFPC, except that nodes in occurs lists now also have back pointers (see Figure 3.8): if node C is in N.occurs[A] for some node N and atom A, then the back pointer C.occurs^{-1}[A] allows directly accessing this list item (i.e., in constant time) from C. An internal node keeps its back pointers in an array indexed by atoms (for a leaf, the only atom of interest is that of its label). To further simplify the back pointers, we require that for any particular atom, any node should be in at most one occurs list. This will ensure that, from any particular node, there is at most one occurs back pointer for any particular atom. It can be verified that this constraint is enforced by the procedures presented so far; we will continue to enforce it for any new procedures that are presented. In effect, we add it explicitly to the occurs invariant:

Addition to Occurs invariant: For any atom A and any node C, there is at most one node N that is accessible from the Root using Occurs[A] links such that C is in N.occurs[A].

Creating and maintaining these back pointers does not add to the asymptotic complexity of the algorithm, since these operations are performed along with the operations on occurs lists with only a constant-factor overhead. Since access of back pointers is always driven by some leaf and back pointers are initialized whenever occurs links are set, we never access uninitialized back pointers. Because of these back pointers, MoveLeaf is a constant-time operation, just like ChangeParent.

Procedure MergeOccurs(N,M : NODE) does not explicitly merge the occurs lists of N and M, because this would take time proportional to the total number of propositional symbols, while the subtree at M might be very small. Instead, it sets appropriate links from N's Occurs array to M by traversing the entire subtree at M: for each A-leaf in the subtree, if N.Occurs[A] does not point to anything (i.e., is either Nil or uninitialized [2]) and M.Occurs[A] points to something (i.e., is both non-Nil and initialized), then N is inserted

[2] We will show later how to detect whether an Occurs entry is uninitialized. This will be needed only when factoring rules are introduced, and can be ignored until then.

in M and M.Occurs^{-1}[A] in the occurs[A] chain; inserting N requires changing two entries: M is replaced by N in M.Occurs^{-1}[A], and N.Occurs[A] is set to the singleton list containing M. Note that if we had just set N.Occurs[A] to be M.Occurs[A] in MergeOccurs then the above addition to the Occurs invariant could be violated. MergeOccurs(N,M) calls the recursive procedure RMergeOccurs(N,M,M), whose pseudo-code is:

Procedure RMergeOccurs(N,M,R : NODE):

```
for each C in R.subs do RMergeOccurs(N,M,C);
for each L in R.leafs do {
  A := abs(L.label);
  if (not initialized(N.occurs[A]) or N.occurs[A] = Nil)
      and initialized(M.occurs[A]) and M.occurs[A] ≠ Nil then {
      P := M.occurs^{-1}[A];
      replace M by N in P.occurs[A];
      N.Occurs[A] := Push(M,Nil);
  }
}
```

Note that M.Occurs^{-1}[A] exists, since Root is an A-node for any atom A.

Apart from Collapse, ChangeParent is called only at one place — line 3 of Lift2 with arguments (C,M,N). Since M is deactivated in the next line, it is again not necessary to modify the occurs lists.

Theorem 3.21 (Time complexity of AFPL) *For any input theory Γ, algorithm AFPL takes O(nk) time, where n is the size of tuple $\langle\langle\Gamma\rangle\rangle$ and k is the depth of (the tree representing) Γ.*

Proof: [sketch] Theorem 3.18 shows that AFPC takes O(nk) time. The only extra work in AFPL, as compared to AFPC, is:

1. Some more nodes are pushed into occurs lists (in MoveLeaf at line 4 of Lift1). However, each time a node is pushed into an occurs list, it is also lifted up in the tree. Since this can happen at most nk times, the total number of nodes ever added to the occurs lists remain O(nk).
2. Some more nodes can be pushed into PQ list, during the call to Collapse in line 10 of Lift1. Using the same argument above, the total number of nodes ever added to list PQ remains O(nk).
3. Since L1Q is created from items removed from PQ, the total number of nodes ever added to L1Q is also O(nk). It follows that Lift1 can be called at most O(nk) times.
4. Since L2Q is created from items removed from PQ and the new line added to Simplify, the total number of nodes ever added to L2Q is also O(nk). It follows that Lift2 can be called at most O(nk) times.
5. The call to MergeOccurs(N,M) in Lift2 traverses the entire subtree at M. Since each of these nodes moves one level up in the tree, the total cost of all calls to MergeOccurs(N,M) in Lift2 is O(nk).
6. Each iteration of the for loop in Lift1 lifts a child up the tree. So the total number of iterations across all calls is at most nk.
7. In addition to the costs mentioned above, each call to either Lift1 or Lift2 takes constant time.

Thus, the time complexity remains O(nk). ∎

3.6 AFP: Adding Factoring Rules

We extend the algorithm AFPL for dealing with the factoring rules (F rules). Since the basic idea remains the same, we need a list to keep track of nodes where an F-rule may be applicable. We also need new data

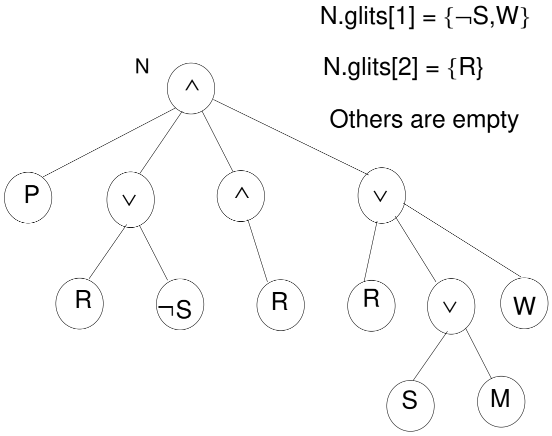

Figure 3.9: Glits lists

structures, namely, the Glits arrays and operations on them, for quickly identifying such nodes. The new algorithm AFP first reduces the theory with respect to all rules except factoring, initializes the new data structures, and then executes the usual loop of applying all the rules. Note that factoring rules are the only rules that require adding new nodes to the tree. We also prove the correctness of AFP and provide a quadratic bound on its running time.

The following list keeps track of potential applications of the F rules:

FQ: list of nodes where F rule may apply

For any node N, a literal α can be factored out of the children of N iff

1. N has at least two non-leaf children,
2. the label of each non-leaf child of N is the dual of N.label, and
3. α labels a child of each non-leaf child of N.

For quick identification of such nodes, we maintain the following information with each internal node N of the tree which has at least one non-leaf child:

Glits: an array such that for any x (from 1 through the number of leaf grandchildren of N), N.glits[x] is the list of literals that label exactly x grandchildren of N whose parents are labeled with the dual of N.label.

Glits lists keep track of literals that label the leaf grandchildren of a node. This information is also maintained in nodes with only one non-leaf child since they can get more children by using Rule L2. Since F rule never applies to the Root, we don't need a glits array for the Root. Figure 3.9 shows an example of glits lists.

The following invariants ensure that lists glits have the correct information and that list FQ has all the nodes where F rules are applicable:

Glits Invariant: For any internal node N, any literal α, and any positive integer x: α is in list N.glits[x] iff N has x grandchildren labeled by α whose parents are labeled with the dual of N.label.

FQ Invariant: For any internal node N, if Count(N.subs) > 1 and N.glits[Count(N.subs)] $\neq$ Nil then N is in list FQ.

Since the queue FQ is accessed only when no more propagation steps can be applied (because PQ is empty and the Occurs invariant holds), there cannot be any node with two or more children labeled by the same literal. N.glits[x] is therefore equal to the list of literals that label a leaf-child of exactly x children of N labeled with the dual of N.label.

Once established, these invariants continue to hold at all times, except for certain lines of the code — such violations are mentioned explicitly.

Some more procedures are used in the description of AFP:

1. Procedure **SetGcount(N,α,x)** adds literal α to the list N.glits[x], if it is not already there.
2. Procedure **IncGcount(N,α)** removes α from some glits list of N, say N.glits[x], and adds it to to the list with the next higher index, i.e., N.glits[x+1]. If α is not in any glits list of N then x is considered to be 0. Also, if Count(N.subs) = x+1 > 1 then N is added to the list FQ.
3. Procedure **DecGcount(N,α)** removes α from some glits list of N, say N.glits[x], and adds it to the list with the next lower index, i.e., N.glits[x-1].

By using additional data structures, we will later show how each of the above procedures can be made to run in constant-time.

AFP first calls AFPL for reading the input theory and reducing it to a normal form with respect to S, P, and L rules. It then calls InitFactor to set the glits arrays and the list FQ. As before, nodes are then removed from the lists PQ, L1Q, L2Q, and FQ (in that order) and the corresponding rules are applied if they are still applicable.

Procedure AFP:

```
/* reads the theory and rewrites it to an irreducible form
PRE: input I correctly encodes a theory, say Γ
POST: theory Δ represented by the tree is irreducible
      Γ ⇒!_FP Δ
      all invariants are satisfied */
AFPL; /* reads Γ and reduces it using S,P, and L rules */
InitFactor; /* set glits and FQ */
loop {
  if PQ ≠ Nil then { /* possible P rule */
      Propagate(Head(PQ));
      AddQ(Pop(PQ),L1Q);
  } elseif L1Q ≠ Nil then { /* possible L1 rule */
      Lift1(Head(L1Q));
      AddQ(Pop(L1Q),L2Q);
  } elseif L2Q ≠ Nil then { /* possible L2 rule */
      Lift2(Head(L2Q));
      Pop(L2Q);
  } elseif FQ ≠ Nil then { /* possible F1 rule */
      Factor(Head(FQ));
      Pop(FQ);
  } else exit; /* got an irreducible form */
}
```

end (FP).

InitFactor sets the glits arrays and FQ list so that the corresponding invariants are established. It makes two passes over the tree: the first pass (InitGlits) is used to initialize all the relevant fields of glits arrays;

only these initialized fields are accessed in the second pass (SetGlits). As we shall see later, not requiring the initialization of all the glits arrays will be important in our analysis of the time complexity of AFP.

Procedure InitFactor

```
/* initializes and then sets the glits fields
PRE: list PQ is empty
POST: also, Glits and FQ Invariant */
  FQ := Nil;
  For each N in Root.subs do InitGlits(N);
  For each N in Root.subs do SetGlits(N);
```

end (InitGlits).

InitGlits(N) recursively traverses the subtree at N, setting the relevant glits fields to Nil. It also collects all the relevant literals in the glits[0] fields. More precisely, it establishes the following assertion that is used by SetGlits:

Glits intermediate-assertion: For any internal node N

1. if N has at least x ($\neq 0$) non-leaf children then N.glits[x] = Nil;
2. for any literal α: the subtree at N has a leaf labeled by α iff α is in list N.glits[0].

Procedure InitGlits (N : NODE)

```
/* recursively initializes the glits fields
PRE: N is an internal node
POST: Glits intermediate-assertion for the subtree at N */
  for each L in N.leafs do { /* initialize glits[0] for this leaf */
    P := N.parent; /* P walks up to the Root */
    while P ≠ Root do {
        SetGcount(P,L.label,0); /* add L.label to P.glits[0] */
        P := P.parent;
    }
  }
  x := 0; /* counter of non-leaf children */
  N.glits[x++] := Nil;
  for each C in N.subs do {
    N.glits[x++] := Nil;
    InitGlits(C);
  }
```

end (InitGlits).

SetGlits(N) also recursively traverses the subtree at N. For each leaf with label, say α, encountered, it increments the index of α in the glits array of its grandparent. Since the tree is irreducible with respect to P rules, each node has at most one leaf labeled by α. If an F rule is applicable at node N then it is added to the list FQ.

Procedure SetGlits (N : NODE)

```
/* recursively sets the glits fields
PRE: N is an internal node;
      list PQ is empty;
      Tree satisfies Glits intermediate-assertion
POST: Glits and FQ Invariants for the subtree at N */
  for each C in N.subs do {
    if N.label ≠ C.label then
```

```
        for each L in C.leafs do
            IncGcount(N, L.label);
    SetGlits(C);
}
if Count(N.subs) > 1 and N.glits[Count(N.subs)] ≠ Nil then
    AddQ(N, FQ);
```

end (InitGlits).

We also make some changes to the algorithms of the previous section, for maintaining the Glits and FQ invariants. These changes are applicable only after the call to AFPL is completed; a boolean tag can be used to indicate this.

For any internal node N:

1. after a node is removed from N.subs, if Count(N.subs) > 1 and N.glits[Count(N.subs)] is non-Nil then N is added to list FQ;
2. before a node is added to N.subs, N.glits[Count(N.subs)+1] is initialized to Nil.

Any non-recursive call to DeleteTree(N), i.e., a call from outside DeleteTree, is replaced by a call to NewDeleteTree(N) for updating glits affected by the nodes that are removed:

Procedure NewDeleteTree (N : NODE)

```
P := N.parent;
if P ≠ Root then
    if Leaf?(N) then
        if (G := P.parent) ≠ Root and P.label ≠ G.label then
            DecGcount(G,N.label)
    elseif N.label ≠ P.label then
        for each L in N.leafs do
            DecGcount(P, L.label);
body of DeleteTree(N);
```

end (NewDeleteTree).

Procedures MoveLeaf and ChangeParent are also modified, as shown below, so that glits arrays are maintained correctly. The operations on glits in **MoveLeaf(C,N,P)** are:

```
if (G := P.parent) ≠ Root and P.label ≠ G.label
    then IncGcount(G, C.label);
if (R := N.parent) ≠ Root and N.label ≠ R.label
    then DecGcount(R, C.label);
```

The corresponding operations on glits in **ChangeParent(C,N,P)** are:

```
if Leaf?(C) then {
    if (G := P.parent) ≠ Root and P.label ≠ G.label
        then IncGcount(G, C.label);
    if (R := N.parent) ≠ Root and N.label ≠ R.label
        then DecGcount(R, C.label);
} else {
    if P ≠ Root and C.label ≠ P.label then
        for each L in C.leafs do
            IncGcount(P, L.label);
```

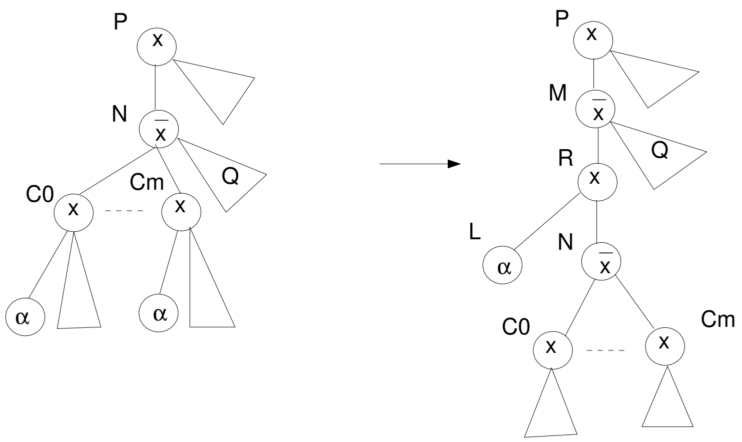

Figure 3.10: Factoring

```
   if N ≠ Root and C.label ≠ N.label then
       for each L in C.leafs do
           DecGcount(N, L.label);
}
```

Once a node is created, any change in its position in the tree is made by calling one of the above three procedures (or by the procedure Factor which is described later). Thus, the above modifications ensure that the Glits invariant is satisfied.

Factor(N) applies F rules as many times as possible to the subtree at node N (Figure 3.10). Two new nodes, M and R, are created and inserted between N and its parent in the tree. All leaves of N are moved to M and all factored literals are made the children of R. N is left with only non-leaf children from which the factored leaves have been removed. These leaves are removed using calls to Simplify that change their labels to appropriate truth-constants and reduce them with respect to S rules. If N did not have any leaf child to begin with, then node M is deactivated by calling Collapse — it is possible to use an L rule after this. As in the case of P rules, we will interleave S rules with an application of F rules.

Procedure Factor (N : NODE)

```
/* checks and applies F rules at Node N
PRE: list PQ is empty
POST: either N is in Pop(FQ) or F rule does not apply at N;
      term(Root_PRE) ⇒*_R term(Root_POST) using F and S rules */
x := Count(N.subs);
if N ≠ Root and x > 1 and N.glits[x] ≠ Nil then {
            /* apply factoring */
  P := N.parent;
  val := (N.label == ∨); /* deleted leaves are substituted by val */
  M := CreateNode(N.label,Nil); SetParent(M,P);
  For each Q in N.leafs do {
      MoveLeaf(Q,N,M);
      Q.pp := M;
  } /* RHS for factoring */
  R := CreateNode(-N.label,Nil); SetParent(R,M);
  ChangeParent(N,P,R);
  M.glits[0] := M.glits[1] := Nil;
```

```
          /* R is the only non-leaf child of M */
13.     R.glits[0] := R.glits[1] := Nil;
14.     while N.glits[x] ≠ Nil do {
          /* apply factoring for each literal */
15.       α := Pop(N.glits[x]); /* literal to be factored */
16.       L := CreateNode(α,Nil); L.parent := R; L.pp := R;
17.       SetGcount(M,α,1);
18.       Push(L,N.Occurs^-1[abs(α)]);
19.       Push(L,R.leafs);
20.       PropAtom(α, val, N,L);
21.       }
22.     }
23.     if M.leafs = Nil then {
24.       Collapse(M);
25.       if R.label = P.label or (R.label = ∧ and P.label = ⊙)
26.         then AddQ(R,L1Q);
27.     }
28. }
```
end (Factor).

Here, Occurs^{-1} in line 18 refers to the occurs back pointer — for any node N and atom A, if N is in S.Occurs[A] for any node S then N.Occurs^{-1}[A] is S. This line ensures that the new leaf L is put in the proper occurs list.

Note that calling Simplify may lead to new nodes being added to the list PQ, collapsing of nodes, etc. Also, the node N may get deactivated during the factoring, or it may get added to the FQ list again. The occurs invariant continues to hold since nodes M and R, which are newly created, are not A-nodes for any atom A. However, these nodes may become A-node because of a subsequent call to Lift2. The relevant occurs entries in these nodes are then set by the call to MergeOccurs.

3.6.1 Correctness of AFP

As before, we first prove the correctness of InitGlits, SetGlits, and Factor, before proving the correctness of AFP.

Proposition 3.22 (Correctness of InitGlits) *For any call of InitGlits(N) during the execution of InitFactor, N is an internal node, and after the call:*

1. *if N has at least x ($\neq 0$) non-leaf children then N.glits[x] = Nil;*
2. *for any literal α: the subtree at N has a leaf labeled by α iff α is in the list N.glits[0].*

Proof: Since N is in the subs list for some node, it must be an internal node. Claim 1 follows directly from line 11 of the procedure. Claim 2 is easily proved by induction on the depth of recursion. The base case, when all leaves in the subtree are children of N, follows from line 4 of the code. The inductive case follows because line 4 adds the child leafs, while the recursive call in line 12 adds the non-child leafs. ∎

Thus, Glits intermediate-assertion is satisfied after all the calls to InitGlits are completed. Also, since AFPL has terminated and the tree is not an exception tree, the list PQ is empty.

Proposition 3.23 (Correctness of SetGlits) *For any call of SetGlits(N) during the execution of InitFactor, N is an internal node, and after the call:*

1. *for any literal α, and any positive integer x: α is in list N.glits[x] iff N has x children that are labeled with dual of N.label and have a leaf-child labeled by α;*

2. *if Count(N.subs) > 1 and N.glits[Count(N.subs)] $\neq$ Nil then N is in list FQ.*

Proof: Since the list PQ is empty, it follows from PQ and pp invariants that for any literal α, each child of N has at most one child labeled by α. Claim 1 is then established by line 4 of the code, while claim 2 is established by line 8. The check in line 2 ensures that C.label is a dual of N.label. ■

It follows that both the Glits and FP invariants are satisfied after the call to InitFactor is completed. Since DeleteTree, Collapse, Lift1, and Lift2 are correct, we only need to show that the two new invariants (Glits and FQ) are not violated by their modified versions. Since the new lines of the code (in NewDeleteTree, MoveLeaf, and ChangeParent) explicitly enforce these invariants, it follows that NewDeleteTree, Collapse, Lift1, and Lift2 are all correct.

Lemma 3.24 (Correctness of Factor) *For any call to Factor(N) during the execution of AFP:*

1. *$term(Root)$ is irreducible with respect to S rules (before and after the call),*
2. *list PQ is empty,*
3. *all invariants are satisfied,*
4. *$term(Root_{PRE}) \Rightarrow_R^* term(Root_{POST})$ using F and S rules, and*
5. *either N is added to the FQ list during the call or the F rule is not applicable at N after the call.*

Proof: From the code of AFP, Factor is called only when PQ is empty. InitFactor establishes Glits and FQ invariants. All other invariants continue to hold after being established by InitPropagate. Since only F and S rules are applied during the call to factor, it follows that $term(Root_{PRE}) \Rightarrow_R^* term(Root_{POST})$ using F and S rules. If N is not added to the FQ list during the call then each literal in the list N.glits[x], where x is the number of non-leaf children of N (before the call), has been factored in the while loop; thus, it follows from the Glits invariant that the F rule is not applicable to N after the call.

We also have to ensure all invariants are satisfied after the call. The only difficulty is with the Glits and Occurs invariants. Glits for P does not change since all the leaf-children of N, the old child of P, are moved to M, the new child of P. The only change in glits is for node N in line 15; the Glits invariant is satisfied since the corresponding leaves are removed in line 20. The new nodes M get a correct glits array in lines 12 and 17, while each list in glits of R is correctly set to Nil in line 13. The occurs invariant is also satisfied since neither R nor M is an A-node for any atom A. ■

Theorem 3.25 (Correctness of AFP) *For any theory Γ, the procedure AFP on input $\langle\!\langle\Gamma\rangle\!\rangle$ returns a tree that represents a theory Δ such that $\Gamma \Rightarrow_{FP}^! \Delta$.*

Proof: Follows directly from the correctness of AFPL, InitFactor, and Factor, and the observation that either an exception tree is returned or all the lists PQ, L1Q, L2Q, and FQ are empty when the algorithm terminates. ■

3.6.2 Complexity of AFP

We prove that the time complexity of AFPC is quadratic in the size of the input theory. Our approach remains the same as that in proving the complexity of AFPC and AFPL, i.e., bound the complexity by the cost of moving each node up the tree all the way to the root. The main difference now is that the level of a node may increase because factoring rules add new nodes to the tree. So, we define factoring depth, which also accounts for the increase due to factoring steps, and show that it is at most linear in the size of the initial tree. We also give details of the new data structures and show how their operations can be performed efficiently.

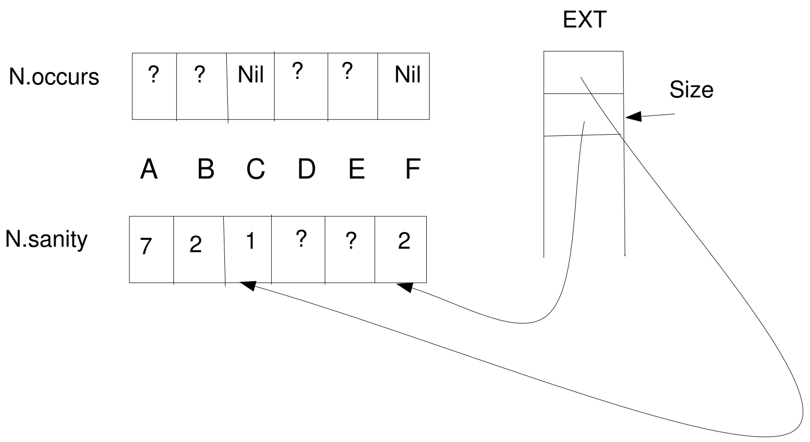

Figure 3.11: Initialization of Occurs arrays of the new nodes created by factoring

Since the new internal nodes introduced by Factor may become A-nodes (for some A) after a call to Lift2, we need a way to initialize the relevant occurs entries in these nodes and distinguish them from the uninitialized entries. This distinction is used in MergeOccurs. For keeping track of initialized occurs entries, we use a global list, EXT, which will have pointers to all these initialized entries, and an array N.Sanity (indexed by atoms) with each new internal node N created by Factor (see [AHU74]). The general idea is that if for some atom A, location N.Occurs[A] is initialized then a new entry is created in EXT that will point back to the initialized location, and N.Sanity[A] will be set to point to this new entry in EXT. To determine whether N.Occurs[A] has been initialized, all we need to do is to verify these two pointers. (We also need to keep track of the size of EXT, so that we do not access uninitialized entries in EXT!)

For example, Figure 3.11 shows the relevant entries after N.occurs[C] and N.Occurs[F] have been initialized. The corresponding entries in N.Sanity refer to indices in the list EXT, whose entries point back to N.Sanity. Uninitialized entries don't have the correct setup of these pointers: N.Sanity[A] is greater than the size of EXT, EXT[N.Sanity[B]] does not point back to N.Sanity[B], and N.Sanity[D] and N.Sanity[E] may not even have integer values. Note that there is only a constant-time overhead for doing this "sanity check" for each access to N.Occurs[A]. Also, the arrays EXT and Sanity need not be initialized to begin with; they get the correct values when the corresponding occurs entries are initialized. Thus, the complexity does not increase because of this check. Note that we do not need this initialization technique for the nodes in the original tree produced by the call to ReadArgs, since InitOccurs initializes all the relevant occurs entries.

For any internal node N, some literal α can be in at most one of the N.glits lists; we maintain an explicit index, indGlits, from literals to this unique occurrence. The initialization issues for the index entries are solved using the same technique that is used for Occurs entries. For example, setting this index would require an additional pass over the tree produced by AFPL. This allows SetGcount, IncGcount, and DecGcount, described in slightly more detail here, to be constant-time operations:

1. Procedure **SetGcount(N : NODE, α : LIT, x : INT)**: if literal α is not in the list N.glits[x], then it is added there and N.indGlits[α] is set to it. Also initialize N.indGlits[α], if not already done.

2. Procedure **IncGcount(N : NODE, α : LIT)**: locate the index x such that α appears in N.glits[x]; this is done through another array, N.indGlits, whose entry N.indGlits[α] would have value x. Remove α from N.glits[x], add it to N.glits[x+1], and update N.indGlits[α] to $x + 1$. Also, if Count(N.subs) = x+1 > 1 then N is added to the list FQ. The pseudo-code for IncGcount is:

```
1. if not initialized(N.indGlits[α]) then SetGcount(N,α,0);
```

```
2. x := N.indGlits[α];
3. move α from N.glits[x] to N.glits[x+1];
4. N.indGlits[α] := x+1;
```

3. Procedure **DecGcount(N : NODE, α : LIT)** is just the reverse of IncGcount, though step 1 is omitted.

Apart from Collapse and Lift2, ChangeParent is called at only one place — line 11 of Factor with argument (N,P,R). Since both the nodes M and R added between P and N are not A-nodes for any atom A, the occurs links from P to N do not need to go through M and R. Thus, as before, there is no need to change any occurs lists. Recall that in contrast, MoveLeaf indeed has to change occurs lists.

Theorem 3.21 showed AFPL(Γ) takes O(nk) time, where n is the size of tuple $\langle\langle\Gamma\rangle\rangle$ and k is the depth of (the tree representing) Γ. We now show that AFP(Γ) takes more time, mainly because of the possible increase in depth of the tree caused by factoring. We first define a new measure of depth that also accounts for the increase due to factoring steps.

Definition 3.6 Any call to Factor(N) is *successful* iff a factoring rule is applicable to the tree rooted at node N. For any execution of AFP(Γ), the *factoring tree* is the unique tree constructed as follows:

1. start from the tree representing Γ;
2. for each successful call to Factor(N) during the execution of AFP(Γ), insert two new nodes between N and its parent.

The *factoring depth* for any execution of AFP(Γ) is the depth of the factoring tree for this execution. ■

Theorem 3.26 shows that the complexity of the algorithm AFP can be expressed in terms of the factoring depth.

Theorem 3.26 *Any execution of AFP(Γ) takes time O(nd), where n is the size of tuple $\langle\langle\Gamma\rangle\rangle$ and d is the factoring depth.*

Proof: (sketch) We have already seen that AFPL costs O(nk) time. We have to account for factoring and the additional rewrite steps that are possible due to factoring. We have to also account for the changes in ChangeParent, since it is no longer a constant-time operation. The additional work to be done in NewDeleteTree is subsumed by the work that is already done in DeleteTree.

Each iteration of the while loop in Factor is one rewrite step using the F rule. Each factoring step can increase the number of nodes by at most 1. Since there can be at most n factoring steps, the total number of nodes at any time is at most 2n.

In the worst case of execution of AFP, each leaf is lifted all the way up to the Root before getting deleted (note that the cost of each of these steps is already accounted for in the above analysis). Even in the new version of ChangeParent (which continues to take constant time, as before), where there is an iteration over the leaves, each of these leaves is moved up in the tree. Since there are at most O(n) leaves and the factoring depth is d = O(n), AFP takes O(nd) time. ■

Since factoring depth depends on the particular execution of the algorithm, we need some bound on factoring depth in terms of the input size. Lemma 3.27 shows that the factoring depth for any execution can be at most linear in the size of the input. This bound is tight: Figure 3.12 shows an example subtree of size O(n), where (n-1) factoring steps lead to an increase in the depth of the subtree by (2n-1); each A and B shown is a distinct atom.

Lemma 3.27 *For any execution of AFP(Γ), the factoring depth is at most (k+2n), where n is the size of tuple $\langle\langle\Gamma\rangle\rangle$ and k is the depth of (the tree representing) Γ.*

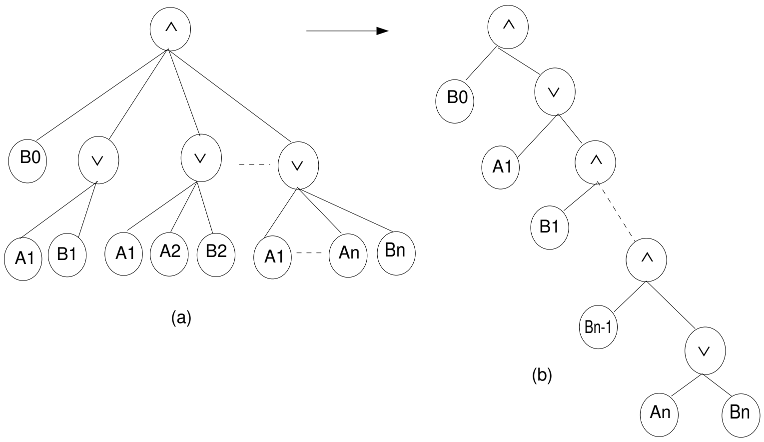

Figure 3.12: Increase in depth due to factoring

Proof: Each factoring step removes at least two leaves from the tree and creates at most one new leaf. Since leaves are never created (outside Factor) after the initial tree is formed, and there are at most n leaves to begin with, it follows that there could be at most n factoring steps. Each factoring step can increase the depth of the tree by 2. Since the initial tree has depth k, the maximum possible factoring depth is (k+2n). ■

It then follows from Lemma 3.27 and Theorem 3.26 that any execution of AFP(Γ) takes O(n^2) time, where n is the size of input tuple $\langle\!\langle\Gamma\rangle\!\rangle$. Thus, the algorithm AFP has quadratic time complexity. Recall that the atoms in Γ are assumed to be integers from 1 to k, where there are k distinct atoms in the theory.

Suppose we restrict our attention to only clausal theories. Since the depth of (the tree representing) any clausal theory is at most 3 and no factoring rule will every apply, the factoring depth for any execution is also at most 3. i.e., a constant. The following is then a corollary of Theorem 3.26:

Corollary 3.28 *For any clausal theory* Γ, *any execution of AFP(*Γ*) takes time* $O(n)$, *where* n *is the size of the tuple* $\langle\!\langle\Gamma\rangle\!\rangle$.

3.7 AFPE: Adding Equality Rules to AFP

We extend the algorithm AFP for dealing with the equality rules (E rules). The E1 rules are used just like S rules, as soon as they become applicable, i.e., whenever an atom of the form $a \doteq a$ for some constant a is encountered while reading the input theory. The E2 rule is applied just like P rules, after queuing the potential applications in a list called EQ, by using an array called Coccurs to locate all literals in which a particular constant appears. There are however some differences:

1. Since the E2 rule propagates an equality atom only if it labels a leaf of the Root, the array Coccurs is needed only for the Root. Contrast this with Occurs arrays which are needed for all non-leaf nodes.
2. Since the E2 rule changes atoms, it is no longer possible to abstract atoms by their codes given by the bijection g of the tuple encoding (see Section 3.3). It will be necessary to keep the structure of an atom around with its code. Note that the leaves are labeled not by atoms but by their codes.
3. Since the E2 rule modifies the atoms in the labels of the leaf nodes, we have to be specially careful in enforcing the Occurs invariant which deals with these labels.

The E2 rule is applied only when the theory is irreducible with respect to P rules. Because of this, whenever an atom of the form $a \doteq b$ is being propagated using the E2 rule $\odot(a \doteq b, B) \Rightarrow \odot(a \doteq b, B[b \overset{*}{\leftarrow} a])$, there will not be any other occurrence of $a \doteq b$ in the tree. Thus, E1 rules never become applicable once the theory has been read and made irreducible with respect to E1 rules, since an atom of the form $a \doteq a$ is never generated thereafter.

The tuple encoding of any theory is extended by explicitly encoding the bijection g at the start. For example, the theory $\Gamma = [\![\vee(\vee(Q, \mathbf{f}), \wedge(P, \vee(\neg P, Q)))]\!]$ is encoded as follows using the bijection $g(P) = 2$ and $g(Q) = 3$:

$$\langle 2, P, Q, 1, -1, 2, -1, 2, 3, -1, 0, 1, 2, 2, -1, 2, -2, 3 \rangle$$

where the first number indicates the number of distinct atoms in Γ which are then explicitly listed, followed by the old encoding.

We have a new data type:

CONST: (individual) constants

We need some new data structures:

Code: a table accessed by atoms (strings), such that Code(A) is the code of atom A.

Atom: an array indexed by codes, such that Atom(n) is the atom whose code is number n.

Coccurs: an array indexed by constants, such that Coccurs[a] is the list of all leaves L such that the constant a appears in the atom Atom[abs(L.label)].

EQ: list of nodes where Rule $E2$ may apply, i.e., the leaves of Root which are labeled by an equality atom of the form $a \doteq b$, where a and b are distinct constants. Since $\doteq$ predicate is symmetric, our use of the notation $a \doteq b$ will implicitly imply that $a \succeq b$.

We will use some more basic operations:

1. Function **Equality?(A : ATOM)** returns true iff A is an equality atom of the form $a \doteq b$, where a and b are distinct constants.
2. Function **IdEquality?(A : ATOM)** returns true iff A is an equality atom of the form $a \doteq a$, where a is a constant.
3. Function **Ancestor(M,R : NODE) : NODE** returns the closest common ancestor of nodes M and R in the tree.

We will maintain some more invariants:

Id Invariant: For any leaf L, IdEquality?(Atom[abs(L.label)]) is false. This invariant ensures that the tree is irreducible with respect to E1 rules.

Code Invariant: for any leaves L and M, if Atom[abs(L.label)] = Atom[abs(M.label)] then abs(L.label) = abs(M.label). This ensures that equality of atoms can be tested using equality of labels.

Coccurs Invariant: For any constant a, Coccurs[a] has exactly all the leaves L such that the constant a appears in the atom Atom[abs(L.label)]. This ensures that all occurrences of a can be accesses through Coccurs[a].

EQ Invariant: If there is a leaf L of Root and constants a and b such that $a \succeq b$, Atom[L.label] = $a \doteq b$, and Coccurs[a] has more than one item then L is in the list EQ. This ensures that all potential applications of E2 rule are kept in list EQ.

Algorithm AFPE is obtained from AFP by adding some new lines and some new procedures. The following lines are inserted just before Line 1 of procedure AFP, for reading the list of atoms and their codes:

```
for i := 1 to read do {
      A := read;
      Code[A] := i+1;
      Atom[i+1] := A;
      }
```

The following lines are added between lines 3 and 4 of procedure ReadFml:

```
IdEquality?(Atom[x]) :  return Tnode;
IdEquality?(Atom[-x]) :  return Fnode;
```

These lines apply the E1 rewrite rules while the input theory is being read. Thus, the tree returned by ReadFml is irreducible with respect to E1 rules, and the Id invariant is satisfied.

The following lines are added between lines 2 and 3 of procedure InitPropagate:

```
EQ := Coccurs := Nil;
SetCoccurs(Root);
```

These lines establish the Coccurs invariant and also initialize the EQ list. The procedure SetCoccurs(N) traverses the subtree rooted at node N and sets the entries in Coccurs for the leaves in the subtree. We do not need two passes, which were required for setting occurs entries, since all slots in Coccurs are initialized (to Nil) before calling SetCoccurs. Note that there is only one Coccurs array for the entire tree.

Procedure SetCoccurs(N : NODE)

```
 /* sets Coccurs array for leaves in the subtree at N
 PRE: Tree and Parent invariants; N is not a leaf;
 POST: also, Coccurs invariant for leaves under N */
  for each L in N.leafs do
    for each constant a in Atom[abs(L.label)] do
        Push(L, Coccurs[a]);
  for each C in N.subs do
    SetCoccurs(C);
```

end (SetCoccurs).

The following line is added between lines 12 and 13 of procedure SetOccurs:

```
if N = Root and Equality?(M.label) then Push(M, EQ)
```

This line adds all potential sites for applying E2 rule to the EQ list. Recall that the E2 rule is said to be applicable to the leaf that is labeled by the equality atom being propagated. Thus, the EQ invariant is satisfied when the call to SetOccurs is completed. Thereafter, whenever the Root acquires a new leaf labeled by an equality atom, the new leaf is added to EQ list (this can happen only after applying S3, L1, or L2 rules).

The following line is added between lines 6 and 7 of procedure AFP:

```
      } elseif EQ ≠ Nil then { PropEq(Pop(EQ));
```

This line, which is in a repeat loop, keeps on popping nodes from the EQ list while calling the procedure PropEq, which applies the E2 rule, if applicable. Note that rule E2 is applied after the P rules, but before the L1, L2, and F1 rules.

When the atom of N is of the form $a \doteq b$ where $a \succeq b$, the procedure PropEq(N) uses the Coccurs array to traverse through all other occurrences of constant a in the tree and replaces them by the constant b. Since these substitutions may make some atoms in the tree equal to one another, the Code invariant requires the same code for them. PropEq may also create some atoms which were not in the input tree, i.e, which do not have any codes assigned to them. These complications are handled in function ChangeLabel, which does the substitution and returns true iff the new atom becomes equal to some other atom in the tree. Various other invariants, which may be violated whenever ChangeLabel returns true, are then re-established.

Procedure PropEq(N : NODE)

```
/* propagate the equality atom of node N
PRE: N is a leaf child of Root;
      Tree is irreducible with respect to P rules;
      all invariants;
POST: E2 rule does not apply at N;
      also, $term(Root_{PRE}) \Rightarrow_R^* term(Root_{POST})$ using the E2 rule *
if Atom[N.label] = $a \doteq b$ where $a \succeq b$ then
  for each M in Coccurs[a] where M $\neq$ N do
      if ChangeLabel(M,a,b) then {
          M.pp := M; /* pp invariant */
          Push(P := M.parent, PQ); /* PQ invariant */
          ChangeOccurs(M); /* Occurs invariant */
          if P $\neq$ Root and (G := P.parent) $\neq$ Root
              and G.label $\neq$ P.label /* Glits, FQ invariants */
              then IncGcount(G,M.label);
          Push(M, Coccurs[b]); /* Coccurs invariant */
      }
  Coccurs[a] := Push(N, Nil);
```

end (PropEq).

Function ChangeLabel substitutes b for a, changing the atom in the label of the leaf M. If B is a new atom then then it is assigned the code of A, which effectively replaces each occurrence of atom A in the tree to B, and false is returned. If A and B are identical, which happens if B was generated earlier as a new atom from A in the same application of the E2 rule, then again false is returned. Otherwise, the code of B becomes the new label of M, modulo the absolute values (which is handled by the variable Neg), and the function returns true.

Function ChangeLabel(M : NODE; a,b : CONST) : BOOL

```
/* change the atom in label of leaf M using E2 rule due to $a \doteq b$
      returns true iff the code of the atom also changes and
      the new atom is not $b \doteq b$.
PRE: M is a leaf; M is in Coccurs[a];
POST: constant a does not appear in Atom[abs(M.label)];
      Code and Atom invariants;
      $term(Root_{POST})$ is obtained from $term(Root_{PRE})$
       using a partial application of the E2 rule */
if M.label < 0 then Neg := true else Neg := false;
A := Atom[abs(M.label)];
B := $A[a \overset{*}{\leftarrow} b]$;
if $A = B$ the return false;
if Code[B] is undefined then {
  Code[B] := Code[A];
  Atom[Code[B]] := B;
  return false;
}
```

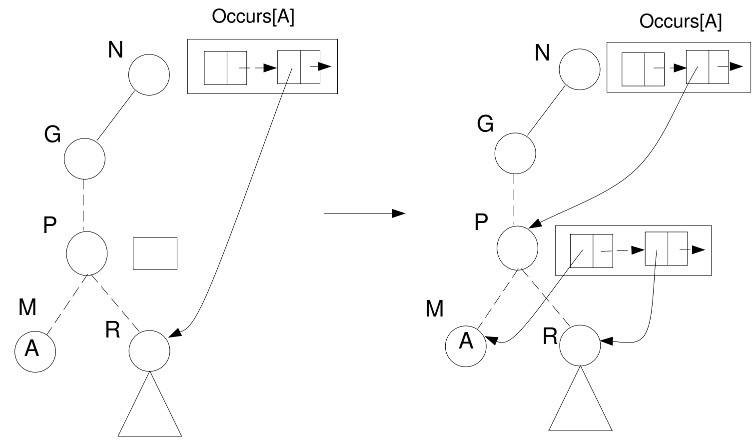

Figure 3.13: ChangeOccurs(M)

```
 10. M.label := Code[B];
 11. if Neg then M.label := - M.label;
 12. return true;
```
end (ChangeLabel).

For example, suppose there are five atoms $a \doteq b$, $a \doteq c$, $P(a, a)$, $P(a, c)$, and $P(b, c)$, which have codes 1, 2, 3, 4, and 5, respectively, in the tree in which the constant a appears and that atom $a \doteq b$ is being propagated using the E2 rule. Suppose PropEq processes leaves in the order M1, M2, M3, and M4, which are labeled by 3, 3, -4, and 2, respectively. ChangeLabel(M1,a,b) assigns the code 3 to $P(b, b)$, which is a new atom. ChangeLabel(M2,a,b) does nothing, since A = B = $P(b, b)$. ChangeLabel(M3,a,b) sets M3.label to be -5. ChangeLabel(M4,a,b) assigns the code 2 to the atom $b \doteq c$, which is a new atom. Of all these calls, only ChangeLabel(M3,a,b) returns true, which indicates that some invariants need to be fixed in PropEq. Even if Root is the parent of M4, there is no need to add M4 to the EQ list; the node should already be in the list, if so required by the EQ invariant.

Since changing constants may lead a non-A-node to become an A-node (for some atom A), re-establishing the occurs invariant needs to be done with care, in procedure ChangeOccurs. Procedure ChangeOccurs(M) traverses up the branch from the leaf M, until it reaches the Root or finds a node N for which occurs[A] is not Nil, where A is the new atom whose code is the label of M. If there is no node between M and N whose subtree has another leaf labeled by A, then M is simply pushed into N.occurs[A]. Otherwise, this node, say P, which was not an A-node becomes an A-node; P.occurs[A] then needs to be properly initialized and set, as shown in Figure 3.13.

The back pointers for the occurs entries, first used in arguing the complexity of AFPL algorithm, are used in searching for the node P. Let G be the child of N whose subtree contains the node M. This subtree can be traversed to find the node R (which is unique because of the occurs invariant) whose back pointer for occurs[A] is N. P is then the closest common ancestor of M and R. Note that P and G may be the same node.

Procedure ChangeOccurs(M : NODE)

```
 /* fix Occurs invariant since the label of leaf M has changed
 PRE: M is a leaf; Occurs invariant holds except for M
 POST: Occurs invariant (Figure 3.13)*/
  1. G := M; N := M.parent; A := abs(M.label);
  2. while N.occurs[A] = Nil and N ≠ Root do {
```

```
 3.   G := N; N := N.parent; }
 4. find R in G's subtree where R.occurs^-[A] = N;
 5. if no such R then Push(M, N.occurs[A]);
 6. else {
 7.   P := Ancestor(M,R);
 8.   replace R by P in N.occurs[A];
 9.   P.occurs[A] = Push(M, Push(R, Nil));
10. }
```
end (ChangeOccurs).

The next theorem shows that AFPE reduces any input theory with respect to the rewrite system FPE. Theorem 3.30 the shows that AFPE is in PTIME.

Theorem 3.29 (Correctness of AFPE) *For any theory Γ, the procedure AFPE on input $\langle\!\langle\Gamma\rangle\!\rangle$ returns a tree that represents a theory Δ such that $\Gamma \Rightarrow^{!}_{FPE} \Delta$.*

Proof: [sketch] The proof, which is based on the invariants, uses the correctness of various procedures and functions called by AFPE. Because of the new lines added to ReadFml, the tree returned by either ReadFml or ReadArgs represents a theory which is irreducible with respect to S and E1 rules and which is obtained from the input theory using these rules. Thus, Id and Code invariants hold when the call to ReadArgs is completed. Because of the new lines added to InitPropagate and SetOccurs, Coccurs and EQ invariants hold when the call to InitPropagate is completed.

After being established, the invariants are violated by the additions to the algorithm only after some call to ChangeLabel completes and returns true. All the violated invariants are then re-established in PropEq. The EQ invariant, which can be violated in the lines of the old algorithm, is also re-established, since any new leaf of Root labeled by an equality atom is added to the list EQ.

The only modification in a theory due to the additions to the algorithm, which happens in ChangeLabel, is sanctioned by the E2 rule. Thus, the output theory can be obtained from the input theory using FPE. All we need to do is to show that the output theory is irreducible with respect to FPE.

If AFPE terminates because of exception, i.e., producing the theory $\emptyset$ or $\{\mathbf{f}\}$, then the output is trivially irreducible with respect to FPE. Otherwise also, the output is irreducible with respect to FPE, since all the lists PQ, EQ, L1Q, L2Q, and FQ are empty when AFPE terminates. Irreducibility with respect to E2 rule is because of the EQ invariant. ∎

Theorem 3.30 (Time complexity of AFPE) *For any input theory Γ, algorithm AFPL takes $O(n^2\ h)$ time, where n is the size of the tuple $\langle\!\langle\Gamma\rangle\!\rangle$ and h is the cost of accessing an item in the table Code.*

Proof: [sketch] It follows from Lemma 3.27 and Theorem 3.26 that any execution of AFP(Γ) takes $\mathrm{O}(\mathrm{n}^2)$ time. Since the cost for AFP was computed using the worst-case scenario, it already includes the cost of applying the non-equality rules that become applicable due to some equality rule (think of having only a single constant in the input theory). Thus, we have to account only for the additional cost in implementing the equality rules.

Reading the new entries in the tuple encoding takes $O(n)$ time. The cost of SetCoccurs is dominated by the cost of line 3, which takes $O(n)$ time. Since there can be $O(n)$ leaves in the initial tree labeled with equality atoms, the number of items ever added to list EQ and the number of times PropEQ is called in also $O(n)$. Each call to ChangeLabel, which costs $O(h)$, changes at least one constant in one atom. Since there can be $O(n)$ distinct constants and $O(n)$ atoms, ChangeLabel can be called $O(n^2)$ times. Thus, the total cost over all calls to ChangeLabel is $O(n^2\ h)$. Since there can be $O(n)$ distinct atoms in the initial tree and each call to ChangeOccurs merges at least two distinct atoms, the total number of calls to ChangeOccurs can be $O(n)$. The cost of each call to ChangeOccurs is dominated by the costs of lines 3 and 4, each of which can take $O(n)$ time. Thus, the total cost of all class to ChangeOccurs is $O(n^2)$.

It follows that the cost of AFPE is at most $O(n^2\, h)$. ∎

Note that if Code is implemented as a hash table of size, say $O(n)$, then h is close to a constant.

3.8 Conclusions

We presented a quadratic time algorithm AFP for reducing theories with respect to FP, and a cubic time algorithm AFPE for reducing theories with respect to FPE. AFP is the only PTIME algorithm we know of, that infers at least as many facts as inferred by CNF-BCP, which is an exponential time algorithm. For some theories, AFP infers more facts than CNF-BCP. Because FP was specified using a rewrite system, we found that converting it to an efficient algorithm was a bit easier, since the task at hand is well defined: find and maintain a list of remaining places where a rule can be applied.

The ability to reorder the application of rewrite rules, because FP is confluent, without concern for changing the results was important and helpful in developing the algorithm and the data structures, and for arguing their correctness by using invariants (e.g., being able to queue some rewrites, while finishing others). Termination also helped in obtaining the tractable algorithm.

Chapter 4

Deductions based on Fact Propagation

4.1 Overview

We have seen that FPE is a relatively efficient (though incomplete) procedure for deducing *facts* from a propositional theory. What if we were interested in deducing more complex formulas, say, clauses? Clearly, FPE by itself provides an effective (though incomplete) way of testing whether a theory is unsatisfiable. We can therefore consider using it as a refutation technique to infer formulas from theories, by defining the following consequence relation: For any theory Γ and any clause ψ:

$$\Gamma \vdash_{\text{FPE}} \psi \qquad \text{iff} \qquad \Gamma \cup [\![\sim\psi]\!] \Rightarrow^*_{\text{FPE}} [\![\mathbf{f}]\!]$$

Recall that FPE is based on propagating facts through theories; since the negation of any clause results in a set of facts, a refutation technique based on FPE might be particularly adept in determining whether a clause can be inferred from a theory. For the rest of the section, we will therefore consider inferring clausal formulas. Remember however, that theories can be in arbitrary form, subject to the usual restriction of being finite.

Although $\Gamma \vdash_{\text{FPE}} \psi$ can be evaluated relatively quickly (time quadratic in the size of Γ and ψ), it is incomplete. There are however some restricted cases in which it is complete:

- When Γ is a Horn theory: Since $\Gamma \cup \{\sim\psi\}$ is a Horn theory, FPE on $\Gamma \cup \{\sim\psi\}$ is just BCP, and thus is complete.
- When clause ψ mentions every atom in Γ: Since $\sim\psi$ corresponds to an interpretation, FPE computes the truth value of Γ.

Let us call ***vivid*** any theory for which $\vdash_{\text{FPE}}$ is complete for inferring clauses. The term "vivid" is inspired by [Lev86], where vivid theories are ones where an answer can be "read off" quickly. For a knowledge base that will be accessed frequently, it makes sense to consider some kind of a "compilation" process that finds a logically equivalent vivid theory. (In fact, even approximate vivid knowledge bases are of interest, as illustrated in [SK91].) It turns out that such a compilation is possible for our $\vdash_{\text{FPE}}$ relation: we present a function Viv, defined in terms of lattice-theoretic fixed-points, such that for every Γ there is a k for which $\text{Viv}(\Gamma, k)$ is vivid. The obvious algorithm that computes $\text{Viv}(\Gamma, k)$ runs in time polynomial in the size of Γ, but exponential in k.[1]

Some theories require higher values of k to be made vivid, others lower. Since Viv turns out to be monotonic in both its arguments, let us call the lowest value k for which $\text{Viv}(\Gamma, k)$ is vivid to be the *intricacy*

[1] A strict interpretation of "reading off" quickly the answers of clausal queries would require a vivid knowledge base to explicitly contain all the prime implicants [Rd87]: a clause is then entailed iff the knowledge base contains a subclause of the clause. The problem with this approach is that vivifying even some Horn theories, which are already vivid according to our definition, leads to an exponential blow-up in their sizes. Note however that any knowledge base that is vivid using the strict interpretation is also vivid using our definition.

of Γ. As one would expect, the intricacy of some theories is proportional to the size of the theory, so making them vivid takes time exponential in their size. There are, however, families of theories (e.g. the 2-CNF theories) for which intricacy can be proven to be bounded by a fixed constant.

We also develop an alternate characterization of $\mathrm{Viv}(\Gamma, k)$ in terms of a family of increasingly complete consequence relations for limited inferencing. This is based on the observation that a source of incompleteness in $\vdash_{\text{FPE}}$ is its inability to use previously inferred clauses for inferring new clauses. For example, for the theory $\Gamma = \{(P \vee Q), (P \vee \neg Q), (\neg P \vee R \vee S), (\neg P \vee R \vee \neg S)\}$, both $\Gamma \vdash_{\text{FPE}} P$ and $\Gamma \cup \{P\} \vdash_{\text{FPE}} R$, but $\Gamma \nvdash_{\text{FPE}} R$. In other words, while P can be inferred from Γ, and R can be inferred if P is added to Γ, R can't be inferred from Γ itself. The following inference system defines a consequence relation $\vdash$ obtained by adding this capability to $\vdash_{\text{FPE}}$:

$$1. \quad \frac{\Gamma \vdash_{\text{FPE}} \psi}{\Gamma \vdash \psi}$$

$$2. \quad \frac{\Gamma \vdash \psi; \qquad \Gamma, \psi \vdash \varphi}{\Gamma \vdash \varphi}$$

The consequence relations $\vdash$ is indeed complete. Unfortunately, it is therefore also intractable.

We show that by restricting the size of ψ in Rule 2 of $\vdash$, we obtain tractable consequence relations that are more complete than $\vdash_{\text{FPE}}$. For example, restricting ψ to be a unit clause provides a tractable consequence relation that is complete for 2-CNF theories, which $\vdash_{\text{FPE}}$ is not. The following inference system defines this family $\vdash_k$ of consequence relations, where k is any natural number:

$$1. \quad \frac{\Gamma \vdash_{\text{FPE}} \psi}{\Gamma \vdash_k \psi}$$

$$2. \quad \frac{\Gamma \vdash_k \psi; \qquad \Gamma, \psi \vdash_k \varphi}{\Gamma \vdash_k \varphi} \quad \text{for } |\psi| \leq k$$

We will show that for any number k, $\vdash_k$ is a sound, monotonic, and tractable consequence relation, which is incomplete. The completeness of $\vdash_k$ increases with k: for any Γ, ψ, and k, if $\Gamma \vdash_k \psi$ then $\Gamma \vdash_{k+1} \psi$. Note that $\vdash_0$ is identical to $\vdash_{\text{FPE}}$.

We will show the following relation between the function Viv and the consequence relation $\vdash_k$: the set of clauses inferable from $\mathrm{Viv}(\Gamma, k)$ using $\vdash_{\text{FPE}}$ is exactly the set of clauses inferable from Γ using $\vdash_k$. Hence, the intricacy of any theory Γ turns out to be the least k for which $\vdash_k$ is complete.

For any (finite) theory Γ, there is a natural class of clauses, called basic clauses, that are built from the predicates and constants in Γ such that logical constants and repetition of atoms are not allowed in a clause. We show that $\vdash_{\text{FPE}}$ from any theory is complete for clauses iff it is complete for its basic clauses. For all other clauses, either $\vdash_{\text{FPE}}$ is trivial or it is equivalent to inferring some basic clause. Thus, we mostly restrict our attention to basic clauses.

We have seen in Chapter 2 that the exact details of FPE itself are quite subtle, that the cost of the FPE algorithm could be lower if we were willing to make it weaker than CNF-BCP, and that extensions of FPE may be desired in some situations. To remove this direct dependency on the details of FPE, we will abstract out those properties of rewrite systems that are needed to make our proofs go through into the concept of *admissible* rewrite system; our results then hold for any admissible rewrite system.

4.2 Preliminaries

We continue to use PCE, propositional calculus with equality and generalized connectives, which was introduced in Section 2.3. We will however use the alternative syntax presented in Section 2.3.1 whenever this does not create any confusion. In this section, we give some definitions that will be used later in the chapter. We first define a consequence relation based on the rewrite system FPE. We then define basic clauses that do not allow either logical constants or repetition of atoms, and a way of merging them to produce new basic clauses. We also define collection of basic clauses built using the predicates and constants that appear in a theory. We finally review some definitions and results regarding lattices and fixed-points.

4.2.1 A Consequence Relation

Since the rewrite system FPE, introduced in Section 2.8, is content preserving, it can be used to rewrite a theory into logically equivalent theories. If some rewriting produces an obviously unsatisfiable theory, say $[\![\mathbf{f}]\!]$, then the initial theory must be unsatisfiable. Thus, FPE can be used as a procedure for testing whether a theory is unsatisfiable: a theory is declared unsatisfiable iff it rewrites to $[\![\mathbf{f}]\!]$. (Note that this procedure is incomplete since there are unsatisfiable theories that do not rewrite to $[\![\mathbf{f}]\!]$.) One can therefore use FPE as a refutation technique for inferring clauses from theories, since a clause ψ is entailed by a theory Γ iff the theory $\Gamma \cup [\![\sim\psi]\!]$ is unsatisfiable. By generalizing this observation to any rewrite system R, we define a consequence relation $\vdash_R$ that formalizes the notion of inferring clauses from theories using R:

Definition 4.1 For any rewrite system R, any theory Γ, and any clause ψ, the consequence relation $\vdash_R$ is defined as follows:

$$\Gamma \vdash_R \psi \qquad \text{iff} \qquad \Gamma \cup [\![\sim\psi]\!] \Leftrightarrow_R^* [\![\mathbf{f}]\!]$$

■

For example, consider the theories $\Gamma = [\![(P \vee Q), (\neg P \vee Q), (P \vee \neg Q)]\!]$ and $\Delta = \Gamma \cup [\![(\neg P \vee \neg Q)]\!]$, and clauses $\psi = (P)$ and $\varphi = \mathbf{f}$:

$$\begin{array}{rcll}
\Gamma \cup [\![\sim\psi]\!] & = & [\![(\neg P), (P \vee Q), (\neg P \vee Q), (P \vee \neg Q)]\!] & \\
& \Rightarrow_{FPE} & [\![\neg P, (P \vee Q), (\neg P \vee Q), (P \vee \neg Q)]\!] & \text{(rule } S3_\vee) \\
& \Rightarrow_{FPE} & [\![\neg P, (\mathbf{f} \vee Q), (\mathbf{t} \vee Q), (\mathbf{f} \vee \neg Q)]\!] & \text{(rule } P1_\odot) \\
& \Rightarrow_{FPE}^* & [\![\neg P, (Q), \mathbf{t}, (\neg Q)]\!] & \text{(rules } S2_\vee,\ S1_\vee, \text{ and } S3_\vee) \\
& \Rightarrow_{FPE}^* & [\![\neg P, Q, \neg Q]\!] & \text{(rules } S2_\odot \text{ and } S3_\vee) \\
& \Rightarrow_{FPE} & [\![\neg P, Q, \mathbf{f}]\!] & \text{(rule } P1_\odot) \\
& \Rightarrow_{FPE} & [\![\mathbf{f}]\!] & \text{(rule } S1_\odot)
\end{array}$$

$$\begin{array}{rcll}
\Delta \cup [\![\sim\varphi]\!] & = & [\![(P \vee Q), (\neg P \vee Q), (P \vee \neg Q), (\neg P \vee \neg Q), \mathbf{t}]\!] & \\
& \Rightarrow_{FPE} & [\![(P \vee Q), (\neg P \vee Q), (P \vee \neg Q), (\neg P \vee \neg Q)]\!] & \text{(rule } S2_\odot) \\
& & \text{(irreducible)} &
\end{array}$$

Thus, $\Gamma \vdash_{\text{FPE}} \psi$ and $\Delta \not\vdash_{\text{FPE}} \varphi$, although both $\Gamma \models \psi$ and $\Delta \models \varphi$. It is also easy to verify that if the literal S is added to each clause in Δ, then the logically entailed clause (S) is not inferable using $\vdash_{\text{FPE}}$. The second example above shows that $\vdash_{\text{FPE}}$ is not complete. There are however some restricted cases in which it is complete:

1. Inferring clauses from Horn theories: Consider any Horn theory Γ and any clause ψ. Since $\sim\psi$ rewrites to a set of facts, $\Gamma \cup \{\sim\psi\}$ rewrites to a Horn theory. Since Clausal BCP is identical to Horn Pebbling for Horn theories (see Section 2.5) and Horn theories do not have the $\doteq$ predicate, it follows from Theorem 2.18 and Proposition 2.19 that FPE on $\Gamma \cup \{\sim\psi\}$ is just Horn Pebbling, which is known to be complete for Horn theories.[DG84]

2. Inferring clauses that mention every atom in the theory: Consider any theory Γ and any clause ψ such that atoms$(\Gamma) \subseteq$ atoms(ψ). In this case $\sim\psi$ is either inconsistent or corresponds to an interpretation, in which FPE computes the truth value of Γ. In the latter case, the completeness claim follows from Theorem 4.1 and Proposition 4.6, which are stated and proved later.

3. Inferring clauses from positive theories: Consider any positive theory Γ and any clause ψ. Since $\sim\psi$ corresponds to a set of facts, FPE propagates these through the clauses of Γ, resulting in a theory

Γ' which contains only facts and positive formulas, with no common literals between facts and other formulas. Such a theory is always satisfiable unless it contains $\mathbf{f}$, in which case $\Gamma' = \{\mathbf{f}\}$. Therefore $\vdash_{\text{FPE}}$ is complete for inferring clauses from positive theories. A symmetric argument applies for theories with only negative clauses.

4. Inferring clauses from *satisfiable* 2-CNF theories: Consider any satisfiable 2-CNF theory Γ and any clause ψ. As above, FPE keeps propagating literals, so that every original 2-clause is either unchanged or reduces to a fact (a single literal or logical constant). Therefore, once again $\Gamma \cup \{\sim\psi\}$ reduces to a theory Γ' which contains only facts and some formulas having no common literals with facts. But these formulas form a subset of Γ, and since Γ was originally satisfiable, the subset of it is also satisfiable. Therefore $\Gamma \cup \{\sim\psi\}$ is unsatisfiable iff $\Gamma' = \{\mathbf{f}\}$. Hence $\vdash_{\text{FPE}}$ is complete for inferring clauses from satisfiable 2-CNF theories.

Theories for which $\vdash_R$ is sound and complete are called R-vivid:

Definition 4.2 For any rewrite system R, a theory Γ is called <u>*R-vivid*</u> iff for any clause ψ: $\Gamma \models \psi$ iff $\Gamma \vdash_R \psi$. ∎

Since any theory can be reduced with respect to FPE in time cubic in the size of the theory (Theorem 3.30), $\Gamma \vdash_{\text{FPE}} \psi$ can also be tested in time cubic in the size of Γ and ψ. If Γ and ψ do not contain the $\doteq$ predicate, then $\Gamma \vdash_{\text{FPE}} \psi$ can be tested in even quadratic time (Theorem 3.26). Thus, it is relatively efficient to determine whether a clause is entailed by a FPE-vivid theory.

4.2.2 Basic Clauses

We now define a restriction on clauses. Recall that a clause is a disjunction of zero or more facts, i.e., logical constants and literals. Any occurrence of the fact $\mathbf{t}$ or occurrences of complementary literals in a clause makes it logically equivalent to the fact $\mathbf{t}$. Occurrence of the fact $\mathbf{f}$ or duplicate occurrences of literals in a clause can be removed to produce a logically equivalent clause. These observations motivate the notion of basic clauses which are clauses that do not allow either logical constants or repetition of atoms:

Definition 4.3 A <u>*basic clause*</u> is a clause $(\psi_1 \vee \ldots \vee \psi_n)$ (where $n \geq 0$) such that

1. each ψ_i $(i \in 1 \ldots n)$ is a literal;
2. for each $i, j \in 1 \ldots n$, if $i \neq j$ then $\psi_i \neq \psi_j$ and $\psi_i \neq \sim\psi_j$.

n is referred to as the <u>*size*</u> of the basic clause. ∎

For example, while $(P \vee Q)$ is a basic clause, neither $(P \vee P)$ or $(P \vee \neg P)$ are basic clauses. Note that $\mathbf{f}$ is the only basic clause of size 0. We use symbols μ, π, σ, etc. to denote basic clauses and symbols Π, Σ, etc. to denote clausal theories containing only basic clauses.

We need some way to merge basic clauses to produce larger basic clauses. Informally, merging basic clauses produces a logically weaker basic clause by taking a union of their literals. To avoid producing non-basic clauses, basic clauses may be merged only if they do not contain complementary literals. Basic clausal theories are merged by merging each combination of clauses selected from each theory.

Definition 4.4 For any $n \geq 2$, basic clauses $\mu_1, \ldots, \mu_n$ are <u>*compatible*</u> iff there is no atom p such that p occurs in μ_i and $\neg p$ occurs in μ_j for some $i, j \in 1 \ldots n$. If $\mu_1, \ldots, \mu_n$ are compatible basic clauses then their <u>*merger*</u> $\mu_1 \stackrel{\circ}{\vee} \ldots \stackrel{\circ}{\vee} \mu_n$ is the unique (disregarding order of literals) basic clause μ such that for any literal α, α is a literal in μ iff α is a literal in μ_i for some $i \in 1 \ldots n$. For any $n \geq 2$ and any basic clausal theories $\Gamma_1, \ldots, \Gamma_n$, the theory $\Gamma_1 \stackrel{\circ}{\vee} \ldots \stackrel{\circ}{\vee} \Gamma_n$ is given by

$$\left[\!\left[\mu_1 \stackrel{\circ}{\vee} \ldots \stackrel{\circ}{\vee} \mu_n \mid \forall i \in 1 \ldots n, \mu_i \in \Gamma_i \text{ and } \mu_1, \ldots, \mu_n \text{ are compatible}\right]\!\right]$$

∎

For example, consider the basic clauses $\mu_1 = (P \vee Q)$ and $\mu_2 = (P \vee \neg R)$. Since the two clauses are compatible, their merger $\mu_1 \overset{\circ}{\vee} \mu_2$ is the basic clause $(P \vee Q \vee \neg R)$. Note that basic clauses $(P \vee Q)$ and $(\neg P \vee R)$ can't be merged since they are not compatible.

4.2.3 Herbrand Bases

Given any theory, we will now define the collection of atoms and basic clauses that can be constructed from the symbols in the theory. The collection of atoms is called the Herbrand base and the collection of basic clauses is called the extended Herbrand base of the theory. For defining these terms, we need some notation for referring to the building blocks of formulas and theories:

Definition 4.5 For any formula ψ:

1. $\mathrm{atoms}(\psi) = \{p \mid p$ is an atom and either p or $\neg p$ is a subformula of $\psi\}$;
2. $\mathrm{lits}(\psi) = \{p, \neg p \mid p \in \mathrm{atoms}(\psi)\}$;
3. $\mathrm{consts}(\psi) = \{a \in \mathcal{C} \mid \exists p \in \mathrm{atoms}(\psi)$ *s.t.* a is an argument of $p\}$;
4. $\mathrm{preds}(\psi) = \{P \in \mathcal{P} \mid \exists p \in \mathrm{atoms}(\psi)$ *s.t.* P is the predicate of $p\}$.

For any theory Γ:

1. $\mathrm{atoms}(\Gamma) = \cup\{\mathrm{atoms}(\psi) \mid \psi \in \Gamma\}$;
2. $\mathrm{lits}(\Gamma) = \cup\{\mathrm{lits}(\psi) \mid \psi \in \Gamma\}$;
3. $\mathrm{consts}(\Gamma) = \cup\{\mathrm{consts}(\psi) \mid \psi \in \Gamma\}$;
4. $\mathrm{preds}(\Gamma) = \cup\{\mathrm{preds}(\psi) \mid \psi \in \Gamma\}$.

■

For example, if $\psi = (P \wedge (\neg P \vee Q) \wedge \neg S)$ then $\mathrm{atoms}(\psi) = \mathrm{preds}(\psi) = \{P, Q, S\}$, $\mathrm{lits}(\psi) = \{P, \neg P, Q, \neg Q, S, \neg S\}$, and $\mathrm{consts}(\psi) = \emptyset$. Notice that $\mathrm{lits}(\psi)$ may contain literals that do not occur in ψ, as illustrated by $\neg Q$ in this example. For another example, if $\varphi = ((P \wedge a \doteq b) \vee Q(a, c) \vee \neg Q(b, c))$ then $\mathrm{atoms}(\varphi) = \{P, a \doteq b, Q(a, c), Q(b, c)\}$, $\mathrm{consts}(\varphi) = \{a, b, c\}$, and $\mathrm{preds}(\varphi) = \{\doteq, P, Q\}$.

For any particular theory, its Herbrand base, as usual, is the set of all atoms that could be constructed from the predicates and literals that appear in the theory:

Definition 4.6 For any theory Γ, the *Herbrand base* is defined to be the set $\mathrm{HB}(\Gamma) = \{P(a_1, \ldots, a_n) \mid n \geq 0; P$ is an n-place predicate in $\mathrm{preds}(\Gamma); a_1, \ldots, a_n \in \mathrm{consts}(\Gamma)\}$. ■

For example, if $\Delta = [\![P(a, b), Q(a)]\!]$ then $\mathrm{preds}(\Delta) = \{P, Q\}$, $\mathrm{consts}(\Delta) = \{a, b\}$, and the Herbrand base of Δ is:

$$\mathrm{HB}(\Delta) = \{P(a, a), P(a, b), P(b, a), P(b, b), Q(a), Q(b)\}$$

As this example shows, the special equality predicate $\doteq$ is not used in constructing the Herbrand base if the theory does not contain any formula involving $\doteq$.

Note that if $\mathrm{consts}(\Gamma)$ is empty then $\mathrm{HB}(\Gamma) = \mathrm{atoms}(\Gamma) = \mathrm{preds}(\Gamma)$. Further, for any theories Γ and Δ, if $\Gamma \subseteq \Delta$ then $\mathrm{HB}(\Gamma) \subseteq \mathrm{HB}(\Delta)$.

The collection of all basic clauses built using the atoms in the Herbrand base of a theory will be called the extended Herbrand base of the theory. The extended Herbrand base is constructed in layers, by restricting the size of the basic clauses in each layer.

Definition 4.7 For any set $\mathcal{A}$ of atoms and any $k \in \mathcal{N}$, a basic clause μ is called a <u>*k-clause over $\mathcal{A}$*</u> iff atoms$(\mu) \subseteq \mathcal{A}$ and the size of μ is at most k. ∎

It follows that $\mathbf{f}$ is a 0-clause over any set of atoms. Note that the size of a k-clause may be less than k; for example, $(P \vee Q)$ is a k-clause for each $k \geq 2$. We now define collections of k-clauses built using the atoms in the Herbrand base of a given theory.

Definition 4.8 For any theory Γ and any $k \in \mathcal{N}$, the <u>*k-extended Herbrand base*</u> of Γ is the set:

$$E(\Gamma, k) = \{\mu \mid \ \mu \text{ is a } k\text{-clause over HB}(\Gamma)\}$$

For any theory Γ, the <u>*extended Herbrand base*</u> of Γ is the set:

$$E(\Gamma) = \cup\{E(\Gamma, k) \mid k \in \mathcal{N}\}$$

∎

For example, if $\Gamma = \{(P \vee Q), (\neg P \vee Q), (P \vee \neg Q)\}$ then:

$$\begin{aligned}
\text{HB}(\Gamma) &= \{P, Q\} \\
E(\Gamma, 0) &= \{\mathbf{f}\} \\
E(\Gamma, 1) &= E(\Gamma, 0) \cup \{(P), (\neg P), (Q), (\neg Q)\} \\
E(\Gamma, k) &= E(\Gamma, 1) \cup \{(P \vee Q), (\neg P \vee Q), (P \vee \neg Q), (\neg P \vee \neg Q)\} \\
& \quad\ \text{(for any } k \geq 2) \\
&= E(\Gamma)
\end{aligned}$$

It follows directly from the definition that $E(\Gamma, k) \subseteq E(\Gamma)$ for any theory Γ and any number k. Also, if $k \geq |\text{HB}(\Gamma)|$ then $E(\Gamma, k) = E(\Gamma)$. Further, for any theory Δ and number p, if $\Gamma \subseteq \Delta$ and $k \leq p$ then $E(\Gamma, k) \subseteq E(\Delta, p)$. Since we are dealing with only finite theories, $E(\Gamma)$ is always finite.

4.2.4 Lattices and Fixed-Points

We now review some definitions and results regarding lattices and fixed-points [BB70]. Consider any binary relation $\preceq$ defined on a finite set S. If $\preceq$ is a partial order, i.e., reflexive, transitive, and antisymmetric, then $(S, \preceq)$ is called a *partially ordered set* (or *poset*). For any $\alpha \in S$ and any $B \subseteq S$: α is called an *upper bound* of B if $\beta \preceq \alpha$ for each $\beta \in B$; α is called an *lower bound* of B if $\alpha \preceq \beta$ for each $\beta \in B$. An upper bound α of B is said to be a *least upper bound (lub)* of B if $\beta \preceq \alpha$ for all upper bounds β of B. Similarly, a lower bound α of B is said to be a *greatest lower bound (glb)* of B if $\alpha \preceq \beta$ for all lower bounds β of B. Poset $(S, \preceq)$ is said to be a *complete lattice* if every subset B of S has both a glb and a lub. For example, the powerset of any finite set is a complete lattice with respect to the subset relation. The relation $\preceq$ is usually dropped when it is clear from the context.

For any complete lattice $(S, \preceq)$, a function $S \rightarrow S$ is also called an *operator* on the lattice. An operator T on a complete lattice S (recall that we usually drop the relation symbol) is *monotonic* if $T(B) \preceq T(B')$ for any subsets B and B' of S such that $B \preceq B'$. For any complete lattice S and any operator T on S, any subset $B \subseteq S$ is a *fixpoint* of T iff $T(B) = B$. The *least fixpoint*, lfp(T), of T is a fixpoint of T such that lfp$(T) \preceq B$ for any fixpoint B of T. In general, an operator may have no fixpoint or no least fixpoint or may have several fixpoints. However, it follows from Tarski [Tar55] that any monotonic operator over a complete lattice has a (unique) least fixpoint.

4.3 Admissible Rewrite Systems

We abstract out those properties of FPE that will be needed for our proofs, and condense them into a set of necessary conditions. This section serves therefore as a repository of results describing the behaviour of admissible rewrite systems, and can be treated as an appendix.

Definition 4.9 A rewrite system R is <u>*admissible*</u> iff R is convergent, content preserving, monotonic with respect to facts, modular, and tractable[2], and satisfies the following properties for any terms s and t, any constants a and b, any literals α's and β's, any bag B of formulas, any clause ψ, any theory Γ, and any number $n \geq 0$:

A†: if $s \Rightarrow_R^* t$ then $\mathrm{preds}(t) \subseteq \mathrm{preds}(s)$ and $\mathrm{consts}(t) \subseteq \mathrm{consts}(s)$;

B†: if s is irreducible with respect to R then either $s = [\![\mathbf{f}]\!]$ or neither $\mathbf{t}$ nor $\mathbf{f}$ is a subterm of s;

C†: $[\![(\alpha_1 \wedge \ldots \wedge \alpha_n)]\!] \Leftrightarrow_R^* [\![\alpha_1, \ldots, \alpha_n]\!]$ (if $n = 0$ then this becomes $[\![\mathbf{t}]\!] \Leftrightarrow_R^* [\![]\!]$;

D†: $[\![\alpha]\!] \cup \Gamma \Leftrightarrow_R^* [\![\alpha]\!] \cup \Gamma[\mathbf{t} \overset{*\sim}{\longleftarrow} \alpha]$;

E†: $\wedge(\mathbf{f}, B) \Leftrightarrow_R^* \mathbf{f}$ and $\vee(\mathbf{t}, B) \Leftrightarrow_R^* \mathbf{t}$ and $\wedge(\mathbf{t}, B) \Leftrightarrow_R^* \wedge(B)$ and $\vee(\mathbf{f}, B) \Leftrightarrow_R^* \vee(B)$;

F†: if the language has the $\doteq$ predicate then $[\![a \doteq b]\!] \cup \Gamma \Leftrightarrow_R^* [\![a \doteq b]\!] \cup \Gamma[b \overset{*}{\longleftarrow} a]$;

G†: if some clause in Γ is a subclause of ψ and $\Gamma \cup [\![\psi]\!] \Leftrightarrow_R^* [\![\mathbf{f}]\!]$ then $\Gamma \Leftrightarrow_R^* [\![\mathbf{f}]\!]$;

H†: if Γ is irreducible with respect to R and B is any consistent bag of literals such that $\Gamma \neq [\![\mathbf{f}]\!]$ and $\mathrm{atoms}(B) \cap \mathrm{atoms}(\Gamma) = \emptyset$ then $\Gamma \cup B$ is irreducible with respect to R;

I†: if the bag of literals in ψ is inconsistent, then $\Gamma \cup [\![\psi]\!] \Leftrightarrow_R^* \Gamma$.

■

Informally, property A† ensures that rewriting a term does not introduce any new constants or predicate symbols. Property B† ensures that all occurrences of $\mathbf{t}$ and $\mathbf{f}$ can be removed by rewriting, except for the theory $[\![\mathbf{f}]\!]$. Property C† ensures that a formula which is a conjunction of literals is rewritten in the same way as all those literals considered as individual formulas. As a special case of C†, when $n = 0$, we have $[\![\mathbf{t}]\!] \Leftrightarrow_R^* [\![]\!]$. Property D† ensures that a literal formula can be propagated through the rest of the theory. Property E† ensures that $\mathbf{t}$ and $\mathbf{f}$ can be simplified in the usual way. Property F† ensures that rewriting does substitute constants by equivalent constants. Property G† ensures that adding to a theory those clauses which have some subclause in the theory has no effect on whether the theory reduces to $\{\mathbf{f}\}$. Property H† ensures that adding a consistent bag of new literals to an irreducible theory produces an irreducible theory. Property I† ensures that adding clauses with complementary literals to a theory produces an equivalent theory with respect to an admissible rewrite system. Recall that since theories are considered as *sets* of formulas, multiple occurrences of formulas in a theory are ignored while using the equivalence relation $\Leftrightarrow_R^*$.

Theorem 4.1 *The rewrite system FPE is admissible.*

Proof: (Sketch) It follows from Theorem 2.40 that FPE is convergent, content preserving, monotonic with respect to facts, and modular with respect to $\odot$, and from Theorem 3.30 that FPE is tractable. Property A† follows from the observation that $\mathrm{preds}(t) \subseteq \mathrm{preds}(s)$ and $\mathrm{consts}(t) \subseteq \mathrm{consts}(s)$ for each rewrite rule instance $s \Rightarrow t$ in FPE. Property B† follows from the observation that some simplification rule applies to any theory, except $[\![\mathbf{f}]\!]$, that has either $\mathbf{t}$ or $\mathbf{f}$ as a subterm. Property C† follows from repeated applications of the lifting rule $L1_\odot$ when $n > 0$; otherwise, it follows from a direct application of the simplification rule $S2_\odot$. Property D† follows from a direct application of the propagation rule $P1_\odot$. Property E† follows directly from the formula simplification rules $S1$ and $S2$. Property F† follows directly from the equality rule $E2_\odot$.

For Property G†, suppose μ is a subclause of clause ψ in Γ, and suppose Γ' be the theory such that $\Gamma = \Gamma' \cup [\![\mu]\!]$, and $\Delta = \Gamma \cup [\![\psi]\!]$. Given that $\Delta \Rightarrow_{\mathrm{FPE}}^* \{\mathbf{f}\}$, we will prove that $\Gamma \Rightarrow_{\mathrm{FPE}}^* \{\mathbf{f}\}$.

[2]Note that tractability is conceptually different from the other requirements for an admissible rewrite system. However, we have included tractability in the definition of admissibility because we will only be interested in (admissible) systems which are also tractable. Thus, rather than saying "tractable and admissible" each time, we can just say "admissible". The intuitions behind the various properties are given after the definition.

If $\Gamma' \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$ or $[\![\mu]\!] \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$ then it follows from modularity and Rule $S1_\odot$ that $\Gamma \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$. Otherwise, suppose $\Gamma' \Rightarrow^!_{\text{FPE}} \Gamma''$. We obtain from modularity that $\Gamma \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\mu]\!]$ and $\Delta \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\mu, \psi]\!]$. Since $\Delta \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$, we obtain from confluence that $\Gamma'' \cup [\![\mu, \psi]\!] \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$. The only way this can happen is that either μ or ψ first reduces to a single literal (say, α). If that literal is from μ then $\Gamma \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$, since ψ is not needed at all. So, suppose that α is obtained from ψ, i.e., $\Gamma'' \cup [\![\psi]\!] \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\alpha]\!]$. Since μ is a subclause of ψ, we obtain using a subsequence of the same rule applications that either $\Gamma'' \cup [\![\mu]\!] \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\alpha]\!]$ or $\Gamma'' \cup [\![\mu]\!] \Rightarrow^*_{\text{FPE}} \Gamma'' \cup \{\mathbf{f}\}$. In the later case, we obtain from Rule $S2_\odot$ that $\Gamma \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$. In the former case, we obtain that $\Delta \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\alpha]\!] \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$. Thus, $\Gamma \Rightarrow^*_{\text{FPE}} \Gamma'' \cup [\![\alpha]\!] \Rightarrow^*_{\text{FPE}} \{\mathbf{f}\}$.

For Property I†, suppose the bag in clause ψ contains α and $\neg\alpha$. We obtain that:

$$\begin{aligned}
\vee(\alpha, \neg\alpha, \ldots) &\Leftrightarrow^*_{\text{FPE}} \vee(\alpha, \mathbf{t}, \ldots) \quad (\text{Rule } P1_\vee)\\
&\Leftrightarrow^*_{\text{FPE}} \vee(\mathbf{t}) \quad (\text{Rule } S1_\vee)\\
&\Leftrightarrow^*_{\text{FPE}} \mathbf{t} \quad (\text{Rule } S3_\vee)
\end{aligned}$$

Using modularity and Rule $S2_\odot$, we then obtain that $\Gamma \cup [\![\vee(\alpha, \neg\alpha, \ldots)]\!] \Leftrightarrow^*_{\text{FPE}} \Gamma$.

Property H† follows since no rules in FPE apply to $\Gamma \cup B$. Thus, FPE is admissible. ∎

FPE is not the only admissible rewrite system; we can similarly prove that the rewrite systems HP, CBCP, and FP for finite PC (i.e., PCE without $\doteq$) are also admissible.[3] Property F† is not relevant for subsets of PCE that do not contain the $\doteq$ predicate, for example, finite PC. In fact, Section 5.5 is the only section in which proofs use Property F†; this property can be ignored for the other sections. Since each of these properties has been given a unique name, we will explicitly refer to precisely these names in the later proofs, instead of referring to the general Theorem 4.1.

Admissible rewrite systems have several additional properties that are of interest to us. Here are some that follow directly from the definitions, where R is any admissible rewrite system, Γ, Γ', and Δ are any theories, and ψ is any clause:

1. The consequence relation $\vdash_R$ is sound, i.e., if $\Gamma \vdash_R \psi$ then $\Gamma \models \psi$. This is because R is content preserving.
2. The consequence relation $\vdash_R$ is invariant under $\Leftrightarrow^*_R$, i.e., if $\Gamma \Leftrightarrow^*_R \Delta$ then $\Gamma \vdash_R \psi$ iff $\Delta \vdash_R \psi$. This follows directly from the modularity of R.

We now present some other properties of admissible rewrite systems. Proposition 4.2 shows that any theory containing $\mathbf{f}$ as a formula (not as a subformula) reduces to the theory $[\![\mathbf{f}]\!]$. Proposition 4.3 shows that the consequence relation $\vdash_R$ is monotonic.

Proposition 4.2 *For any admissible rewrite system R and any theory Γ: $[\![\mathbf{f}]\!] \cup \Gamma \Rightarrow^!_R [\![\mathbf{f}]\!]$.*

Proof: Since R is convergent, there is a unique Δ such that $[\![\mathbf{f}]\!] \cup \Gamma \Rightarrow^!_R \Delta$. Since R is monotonic with respect to facts, facts($[\![\mathbf{f}]\!] \cup \Gamma$) $\subseteq$ facts(Δ), i.e., $\mathbf{f} \in$ facts(Δ). This is possible only if $\mathbf{f}$ is a subterm of Δ, which is irreducible. The result then follows directly from property B†. ∎

It follows from Proposition 4.2 and modularity of R that for any theory Γ and any formula ψ:

1. if $\Gamma \Leftrightarrow^*_R [\![\mathbf{f}]\!]$ then $\Gamma \vdash_R \psi$
2. if $\psi \in \Gamma$ then $\Gamma \Leftrightarrow^*_R [\![\mathbf{f}]\!]$ iff $\Gamma \cup [\![\psi]\!] \Leftrightarrow^*_R [\![\mathbf{f}]\!]$

[3] Property C† is not relevant for HP and CBCP because conjunctions of literals are not allowed as formulas. Regarding Property I† for HP and CBCP, note that clauses containing complementary literals are simply ignored. Moreover, Property C† can be ignored for HP and CBCP because conjunction of literals is not allowed in clausal formulas.

Since the consequence relation $\vdash_R$ is defined in terms of whether a theory reduces to $[\![\mathbf{f}]\!]$, it follows from Claim 2 above that multiple occurrences of formulas in a theory (a bag of formulas) can be removed without changing the consequence relation. We will use this observation to simplify our notation for theories: henceforth, they will be treated as sets of formulas in the rest of this chapter. For example, the theory $[\![\mathbf{f}]\!]$ will now be denoted by $\{\mathbf{f}\}$. We will also abuse the symbol $\Leftrightarrow_R^*$ by ignoring multiple occurrences of formulas. For example, if $\psi \in \Gamma$ then $\Gamma \Leftrightarrow_R^* \Gamma \cup \{\psi\}$.

Since the convergence of R is used in almost all steps of the remaining proofs in this section, we will not explicitly mention it from now on.

Proposition 4.3 *For any admissible rewrite system R, any theories Γ and Δ, and any clause ψ: if $\Gamma \vdash_R \psi$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash_R \psi$.*

Proof: Let Γ' be any theory such that $\Delta = \Gamma \cup \Gamma'$. Since $\Gamma \vdash_R \psi$, we have $\Gamma \cup \{\sim \psi\} \Leftrightarrow_R^* \{\mathbf{f}\}$. Since R is modular, $\Delta \cup \{\sim \psi\} \Leftrightarrow_R^* \{\mathbf{f}\} \cup \Gamma'$. It then follows from Proposition 4.2 that $\Delta \cup \{\sim \psi\} \Rightarrow_R^! \{\mathbf{f}\}$. Thus, $\Delta \vdash_R \psi$. ∎

There are several special settings under which $\vdash_R$ is complete. Proposition 4.4 shows that if a theory has no atomic subformulas, i.e., it is constructed entirely from logical constants and connectives, then it rewrites to either the trivially satisfiable theory $\emptyset$ or the trivially unsatisfiable theory $\{\mathbf{f}\}$. Proposition 4.5 shows that any basic clause in a theory can be inferred using $\vdash_R$. Proposition 4.6 shows that $\vdash_R$ is complete for any clause that contains all the atoms occurring in the theory (i.e., admissible rewrite systems perform truth evaluation). Proposition 4.7 shows that some form of case analysis for literals is possible using $\vdash_R$.

Proposition 4.4 *For any admissible rewrite system R and any theory Γ: if* $\text{atoms}(\Gamma) = \emptyset$ *then either $\Gamma \Rightarrow_R^! \emptyset$ or $\Gamma \Rightarrow_R^! \{\mathbf{f}\}$.*

Proof: Suppose $\Gamma \Rightarrow_R^! \Delta$ and $\Delta \neq \{\mathbf{f}\}$. Since $\text{atoms}(\Gamma) = \emptyset$, we have $\text{consts}(\Gamma) = \text{preds}(\Gamma) = \emptyset$. Using property A†, $\text{consts}(\Delta) = \text{preds}(\Delta) = \emptyset$. Using property B†, neither $\mathbf{t}$ nor $\mathbf{f}$ is a subformula of Δ. This is possible only if there is no formula in Δ, i.e., $\Delta = \emptyset$. ∎

Proposition 4.5 *For any admissible rewrite system R, any theory Γ, and any basic clause μ: if $\mu \in \Gamma$ then $\Gamma \vdash_R \mu$.*

Proof: Suppose $\mu = (\alpha_1 \vee \ldots \vee \alpha_n)$ for some $n \geq 0$, and $\Gamma = \{\mu\} \cup \Delta$ for some theory Δ. We need to show that $\Gamma \cup \{\sim \mu\} \Rightarrow_R^! \{\mathbf{f}\}$. Since $\sim \mu = (\sim \alpha_1 \wedge \ldots \wedge \sim \alpha_n)$, it follows from property C† that $\{\sim\mu\} \Leftrightarrow_R^* \{\sim\alpha_1, \ldots, \sim\alpha_n\}$. Thus,

$$\begin{array}{rcll} \{\sim\mu, \mu\} & \Leftrightarrow_R^* & \{\sim\alpha_1, \ldots, \sim\alpha_n, \mu\} & (R \text{ is modular}) \\ & \Leftrightarrow_R^* & \{\sim\alpha_1, \ldots, \sim\alpha_n, (\mathbf{f} \vee \ldots \vee \mathbf{f})\} & (\text{property D†}) \\ & \Leftrightarrow_R^* & \{\sim\alpha_1, \ldots, \sim\alpha_n, \mathbf{f}\} & (\text{property E†}) \end{array}$$

It then follows directly from Proposition 4.2 and the modularity of R that $\Gamma \cup \{\sim\mu\} \Rightarrow_R^! \{\mathbf{f}\}$. ∎

Proposition 4.6 *For any admissible rewrite system R, any theory Γ, and any basic clause μ: if* $\text{atoms}(\Gamma) \subseteq \text{atoms}(\mu)$ *and $\Gamma \models \mu$ then $\Gamma \vdash_R \mu$.*

Proof: Suppose $\mu = (\alpha_1 \vee \ldots \vee \alpha_n)$ for some $n \geq 0$, $A = \{\sim \alpha_1, \ldots, \sim \alpha_n\}$, and $\Gamma' = \Gamma[\mathbf{t} \hookleftarrow A]$. Since $\sim\mu = (\sim\alpha_1 \wedge \ldots \wedge \sim\alpha_n)$, it follows from property C† that $\{\sim\mu\} \Leftrightarrow_R^* A$. From modularity and property D†, we have $\Gamma \cup \{\sim\mu\} \Leftrightarrow_R^* A \cup \Gamma'$.

Since $\text{atoms}(\Gamma) \subseteq \text{atoms}(\mu)$, we have $\text{atoms}(\Gamma') = \emptyset$. From Proposition 4.4, we have either $\Gamma' \Leftrightarrow_R^* \emptyset$ or $\Gamma' \Leftrightarrow_R^* \{\mathbf{f}\}$. Suppose $\Gamma' \Leftrightarrow_R^* \emptyset$. Using modularity, $\Gamma \cup \{\sim\mu\} \Leftrightarrow_R^* A$. Since A is satisfiable and R is content preserving, it follows that $\Gamma \cup \{\sim\mu\}$ is satisfiable, i.e., $\Gamma \not\models \mu$, a contradiction. Thus, $\Gamma' \Leftrightarrow_R^* \{\mathbf{f}\}$.

Since R is modular, $\Gamma \cup \{\sim\mu\} \Leftrightarrow_R^* A \cup \{\mathbf{f}\}$. Using Proposition 4.2, we have $\Gamma \cup \{\sim\mu\} \Leftrightarrow_R^* \{\mathbf{f}\}$. Thus, $\Gamma \vdash_R \mu$. ∎

Proposition 4.7 *For any admissible rewrite system R, any basic clause μ, and any atom p such that $p \notin$ atoms(μ) , $\{\mu \mathring{\vee} (p), \mu \mathring{\vee} (\neg p)\} \vdash_R \mu$.*

Proof: Since $p \notin$ atoms(μ), both $\mu \mathring{\vee} (p)$ and $\mu \mathring{\vee} (\neg p)$ are basic clauses. Suppose $\Gamma = \{\mu \mathring{\vee} (p), \mu \mathring{\vee} (\neg p), \sim\mu\}$. From property C†, $\{\sim\mu\} \Leftrightarrow^*_R \{\sim\alpha \mid \alpha \text{ is a literal in } \mu\} =$ (say), A.

$$
\begin{array}{rlll}
\Gamma & \Leftrightarrow^*_R & A \cup \{(p \vee \mathbf{f} \ldots \vee \mathbf{f}), (\neg p \vee \mathbf{f} \ldots \vee \mathbf{f})\} & \text{(modularity and Property D†)} \\
& \Leftrightarrow^*_R & A \cup \{(p), (\neg p)\} & \text{(Property E†)} \\
& \Leftrightarrow^*_R & A \cup \{p, \neg p\} & \text{(modularity and Property C†)} \\
& \Leftrightarrow^*_R & A \cup \{p, \mathbf{f}\} & \text{(Property D†)} \\
& \Leftrightarrow^*_R & \{\mathbf{f}\} & \text{(Proposition 4.2)}
\end{array}
$$

Thus, $\{\mu \mathring{\vee} (p), \mu \mathring{\vee} (\neg p)\} \vdash_R \mu$. ■

Finally, we present two technical lemmas concerning equivalences with respect to admissible rewrite systems. Proposition 4.8 shows some cases in which a literal simplifies a theory. Proposition 4.9 shows the use of a clause in simplifying other clauses. These two propositions will be used only in the next chapter.

Proposition 4.8 *For any admissible rewrite system R, any $n \geq 0$, and any literals $\alpha, \alpha_1, \beta_1, \ldots, \alpha_n, \beta_n$:*

1. $\{\alpha, (\sim\alpha \vee \beta_1 \vee \ldots \vee \beta_n)\} \Leftrightarrow^*_R \{\alpha, (\beta_1 \vee \ldots \vee \beta_n)\}$
2. $\{\sim\alpha, ((\alpha \wedge \beta_1) \vee (\alpha_2 \wedge \beta_2) \vee \ldots \vee (\alpha_n \wedge \beta_n))\} \Leftrightarrow^*_R \{\sim\alpha, ((\alpha_2 \wedge \beta_2) \vee \ldots \vee (\alpha_n \wedge \beta_n))\}$

Proof: For any i, suppose

$$
\alpha'_i = \begin{cases} \mathbf{t} & \text{if } \alpha_i = \alpha \\ \mathbf{f} & \text{if } \alpha_i = \sim\alpha \\ \alpha_i & \text{otherwise} \end{cases} \quad \text{and } \beta'_i = \begin{cases} \mathbf{t} & \text{if } \beta_i = \alpha \\ \mathbf{f} & \text{if } \beta_i = \sim\alpha \\ \beta_i & \text{otherwise} \end{cases}
$$

Now,

$$
\begin{array}{rll}
\{\alpha, (\sim\alpha \vee \beta_1 \vee \ldots \vee \beta_n)\} & \Leftrightarrow^*_R & \{\alpha, (\mathbf{f} \vee \beta'_1 \vee \ldots \vee \beta'_n)\} \text{ (Property D†)} \\
& \Leftrightarrow^*_R & \{\alpha, (\beta'_1 \vee \ldots \vee \beta'_n)\} \\
& & \text{(Property E† and modularity)} \\
& \Leftrightarrow^*_R & \{\alpha, (\beta_1 \vee \ldots \vee \beta_n)\} \text{ (Property D†)} \\
\\
\{\sim\alpha, ((\alpha \wedge \beta_1) \vee \ldots \vee (\alpha_n \wedge \beta_n))\} & \Leftrightarrow^*_R & \{\sim\alpha, ((\mathbf{f} \wedge \beta'_1) \wedge (\alpha'_2 \wedge \beta'_2) \vee \ldots \vee (\alpha'_n \wedge \beta'_n))\} \\
& & \text{(Property D†)} \\
& \Leftrightarrow^*_R & \{\sim\alpha, ((\alpha'_2 \wedge \beta'_2) \vee \ldots \vee (\alpha'_n \wedge \beta'_n))\} \\
& & \text{(Property E† and modularity)} \\
& \Leftrightarrow^*_R & \{\sim\alpha, ((\alpha_2 \wedge \beta_2) \vee \ldots \vee (\alpha_n \wedge \beta_n))\} \\
& & \text{(Property D†)}
\end{array}
$$

■

We claim that it follows from Proposition 4.8 that for any basic clause μ and any literals $\alpha_1, \ldots, \alpha_n$ such that $n \geq 0$ and μ is compatible with $(\sim\alpha_1 \vee \ldots \vee \sim\alpha_n)$, we have

$$\{(\alpha_1 \vee \ldots \vee \alpha_n), \mu \mathring{\vee} (\sim\alpha_1), \ldots, \mu \mathring{\vee} (\sim\alpha_n)\} \vdash_R \mu$$

because when the literals of $\sim\mu$ are added to the theory on the left, they are used to rewrite each clause $\mu \mathring{\vee} (\sim\alpha_i)$ to $\sim\alpha_i$, which is then used to obtain $\mathbf{f}$ from the clause $(\alpha_1 \vee \ldots \vee \alpha_n)$. The next result is used in Chapter 5.

Proposition 4.9 *For any admissible rewrite system R, any compatible basic clauses μ and σ, and any basic clausal theory Π,*

$$\{\sim\mu, \sim\sigma\} \cup \Pi \Leftrightarrow_R^* \{\sim(\sigma \overset{\circ}{\vee} \mu)\} \cup (\Pi \overset{\circ}{\vee} \{\mu\})$$

Proof: Using modularity and property C†, we have $\{\sim\mu, \sim\sigma\} \Leftrightarrow_R^* \{\sim(\sigma \overset{\circ}{\vee} \mu)\}$. From property C†, $\{\sim\mu\} \Leftrightarrow_R^* \{\sim\alpha \mid \alpha \text{ is a literal in } \mu\} =$ (say), A.

Consider any $\pi \in \Pi$ such that π and μ are not compatible, i.e., there is a literal subformula α of μ such that $\sim\alpha$ is a subformula of π.

$$\begin{array}{rlll} \{\sim\mu, \pi\} & \Leftrightarrow_R^* & A \cup \{(\mathbf{t} \vee \ldots)\} & \text{(Property D†)} \\ & \Leftrightarrow_R^* & A \cup \{\mathbf{t}\} & \text{(Property E†)} \\ & \Leftrightarrow_R^* & A & \text{(Property C† for } n = 0) \\ & \Leftrightarrow_R^* & \{\sim\mu\} & \end{array}$$

Since this happens for any such π that is not compatible with μ, we have $\{\sim\mu\} \cup \Pi \Leftrightarrow_R^* \{\sim\mu\} \cup \{\pi \in \Pi \mid \pi \text{ is compatible with } \mu\}$.

Now, consider any $\sigma \in \Pi$ such that σ and μ are compatible.

$$\begin{array}{rlll} \{\sim\mu, \sigma \overset{\circ}{\vee} \mu\} & \Leftrightarrow_R^* & \{\sim\mu, \sigma \overset{\circ}{\vee} (\mathbf{f} \vee \ldots \vee \mathbf{f})\} & \text{(Property D†)} \\ & \Leftrightarrow_R^* & \{\sim\mu, \sigma \overset{\circ}{\vee} \mathbf{f}\} & \text{(Property E†)} \\ & \Leftrightarrow_R^* & \{\sim\mu, \sigma\} & \end{array}$$

Combining the above two, we have $\{\sim\mu\} \cup \Pi \Leftrightarrow_R^* \{\sim\mu\} \cup (\Pi \overset{\circ}{\vee} \{\mu\})$. The result then follows from the modularity of R. ∎

We now show that inferring basic clauses using the extended Herbrand base of a theory is the only interesting case; inferring any other clause is either trivial or is equivalent to inferring some basic clause.

Proposition 4.10 *For any admissible rewrite system R, any theory Γ, any literal α, and any bag B of literals:*

1. *if $\alpha \in B$ then $\Gamma \vdash_R \vee(B)$ iff $\Gamma \vdash_R \vee(B, \alpha)$;*
2. *if B is inconsistent, then $\Gamma \vdash_R \vee(B)$;*
3. *otherwise, $\Gamma \vdash_R \vee(B)$ iff $\Gamma \vdash_R \vee(B \cap (\mathrm{HB}(\Gamma) \cup \sim\mathrm{HB}(\Gamma)))$ (i.e., atoms not occurring in Γ can be omitted from B).*

Proof:

1. The "if" direction follows directly from properties C† and G†. The "only if" direction follows directly from Property C† and Proposition 4.3.
2. if B is inconsistent then there is an atom p such that both p and $\neg p$ are in B. Since $\{p, \neg p\} \Rightarrow_R^* \{\mathbf{f}\}$ using Proposition 4.7, it follows from Proposition 4.3 that $\Gamma \cup \{\sim\vee(B)\} \Rightarrow_R^* \{\mathbf{f}\}$. Thus, $\Gamma \vdash_R \vee(B)$.
3. The "if" direction follows directly from Proposition 4.3. For the "only if" direction, let μ denote the basic clause $\vee(B \cap (\mathrm{HB}(\Gamma) \cup \sim\mathrm{HB}(\Gamma)))$ and B' to denote the bag $B - \mathrm{HB}(\Gamma) - \sim\mathrm{HB}(\Gamma)$. Suppose $\Gamma \vdash_R \mu$, i.e., $\Gamma \cup \{\sim\mu\} \Rightarrow_R^! \Delta \neq \{\mathbf{f}\}$. Since $\Gamma \cup \{\sim\vee(B)\} = \Gamma \cup \{\sim\mu, \sim\vee(B')\}$ and $\mathrm{atoms}(B') \cap \mathrm{atoms}(\Gamma \cup \{\sim\mu\}) = \emptyset$, it follows from Property H† that $\Delta \cup \{\sim\vee(B')\}$ is irreducible with respect to R. Thus, $\Gamma \cup \{\sim\vee(B)\} \Rightarrow_R^! \Delta \cup \{\sim\vee(B')\} \neq \{\mathbf{f}\}$, i.e., $\Gamma \vdash_R \vee(B)$.

∎

It follows from Proposition 4.10 that duplicate literals and literals not appearing in a theory can be removed from a clause without effecting the inferability (using $\vdash_R$) of the clause from the theory, and that a clause containing any complimentary literals is inferable from any theory. Thus, all the interesting cases of inferability are covered by considering only the basic clauses constructed from the extended Herbrand base of a theory.

For the rest of the chapter, we will restrict our attention to the admissible rewrite systems only — we will use R to refer to any admissible rewrite system. Moreover, we will use the specific admissible rewrite system FPE for all the examples. Also, theories will be treated as sets of formulas, since multiple occurrences of formulas do not effect the consequence relation $\vdash_R$ based on any admissible rewrite system R.

4.4 A Fixed-point Construction for Viv

For any admissible rewrite system R, theory Γ, and number k, we use $\vdash_R$ to define an operator $T_{R,\Gamma,k}$ on the powerset of the k-extended Herbrand base $E(\Gamma, k)$ of Γ. We show that this operator always has a least fixpoint, denoted by $\text{lfp}(T_{R,\Gamma,k})$ and called the kth fixpoint of Γ (with respect to R), which has several nice properties:

Soundness: each clause in $\text{lfp}(T_{R,\Gamma,k})$ is logically entailed by Γ.

Monotonicity: $\text{lfp}(T_{R,\Gamma,k})$ is monotonic in Γ and k.

Tractability (for small values of k): If $\vdash_R$ is in PTIME then $\text{lfp}(T_{R,\Gamma,k})$ can be obtained in time polynomial in the size of Γ but exponential in k.

Eventual Completeness: if Γ has n distinct atoms then $\text{lfp}(T_{R,\Gamma,n})$ is exactly the set of basic clauses in $E(\Gamma)$ that are entailed by Γ.

We define $\text{Viv}(R, \Gamma, k)$ to be the theory $\Gamma \cup \text{lfp}(T_{R,\Gamma,k})$, and show that for each Γ there is a k for which $\text{Viv}(R, \Gamma, k)$ is R-vivid. The least such value of k is defined to be the R-intricacy of Γ. We show that the FPE-intricacy of any Horn, positive, negative, or satisfiable 2-CNF theory is 0, and the FPE-intricacy of any 2-CNF theory is at most 1.

> *In the remainder of this section, the symbol R will always refer to an arbitrary admissible rewrite system, which will be an implicit part of every definition and theorem, unless otherwise stated.*

4.4.1 The Fixed-Point Construction

The operator $T_{R,\Gamma,k}$ on any set S of k-clauses produces the set of k-clauses that can be inferred from $\Gamma \cup S$ using the consequence relation $\vdash_R$:

Definition 4.10 For any theory Γ and any $k \in \mathcal{N}$, the operator (function) $T_{R,\Gamma,k} : 2^{E(\Gamma,k)} \longrightarrow 2^{E(\Gamma,k)}$ is defined as:

$$T_{R,\Gamma,k}(S) = \{\mu \in E(\Gamma, k) \mid \Gamma \cup S \vdash_R \mu\}$$

where S is any subset of $E(\Gamma, k)$. ∎

Since the powerset of any set is a complete lattice with respect to the subset relation, $(2^{E(\Gamma,k)}, \subseteq)$ is also a complete lattice. Since $E(\Gamma, k)$ is always finite, this complete lattice is also finite. The next lemma shows that the operator $T_{R,\Gamma,k}$ on this lattice is monotonic in its arguments and parameters:

Lemma 4.11 *For any theories Γ and Δ, any $k, p \in \mathcal{N}$, and any subset M of $E(\Gamma, k)$ and S of $E(\Delta, p)$, if $\Gamma \subseteq \Delta$, $k \leq p$, and $M \subseteq S$ then $T_{R,\Gamma,k}(M) \subseteq T_{R,\Delta,p}(S)$.*

Proof: Since $\Gamma \subseteq \Delta$ and $k \leq p$, it follows that $E(\Gamma, k) \subseteq E(\Delta, p)$. Now consider any basic clause μ:

$$\begin{array}{rcl} \mu \in T_{R,\Gamma,k}(M) & \Rightarrow & \mu \in E(\Gamma,k) \text{ and } \Gamma \cup M \vdash_R \mu \quad \text{(definition)} \\ & \Rightarrow & \mu \in E(\Delta,p) \text{ and } \Delta \cup S \vdash_R \mu \\ & & \quad (E(\Gamma,k) \subseteq E(\Delta,p),\ \Gamma \subseteq \Delta,\ M \subseteq S, \text{ and Proposition 4.3}) \\ & \Rightarrow & \mu \in T_{R,\Delta,p}(S) \quad \text{(definition)} \end{array}$$

Thus, $T_{R,\Gamma,k}(M) \subseteq T_{R,\Delta,p}(S)$. ■

Since $T_{R,\Gamma,k}$ is a monotonic operator over a finite lattice, it has a least fixpoint [Tar55], which can also be characterized using the ordinal powers of $T_{R,\Gamma,k}$, defined in the usual manner (c.f. [Llo87]):

Definition 4.11 For any theory Γ and any $k \in \mathcal{N}$, the ordinal powers of the operator $T_{R,\Gamma,k}$ are defined as follows:

$$\begin{array}{rcl} T_{R,\Gamma,k}\uparrow 0 & = & \emptyset \\ T_{R,\Gamma,k}\uparrow n & = & T_{R,\Gamma,k}(T_{R,\Gamma,k}\uparrow(n-1)) \quad (\text{if } n \in \mathcal{N}) \\ T_{R,\Gamma,k}\uparrow \omega & = & \cup\{T_{R,\Gamma,k}\uparrow n \mid n \in \mathcal{N}\} \end{array}$$

■

Using [Tar55], it follows from Lemma 4.11 that the least fixpoint $\mathrm{lfp}(T_{R,\Gamma,k})$ of $T_{R,\Gamma,k}$ is given by $T_{R,\Gamma,k}\uparrow\omega$. We will refer to $\mathrm{lfp}(T_{R,\Gamma,k})$ as the kth fixpoint of Γ; k is said to be the *index* of this fixpoint. The least fixpoint is used to define a function Viv from the set of theories and natural numbers to the set of theories:

Definition 4.12 For any theory Γ and any number k, $\mathrm{Viv}(R, \Gamma, k)$ is defined to be the theory $\Gamma \cup \mathrm{lfp}(T_{R,\Gamma,k})$. ■

Intuitively $T_{R,\Gamma,k}\uparrow 1$ is the set of all k-clauses that can be inferred from Γ alone using fact propagation, $T_{R,\Gamma,k}\uparrow 2$ is the set of all k-clauses that can be inferred from Γ and the clauses in $T_{R,\Gamma,k}\uparrow 1$ using fact propagation, and so on. Note that $\mathrm{Viv}(R, \Gamma, k)$ augments the theory Γ, rather than replacing it, by the theory $\mathrm{lfp}(T_{R,\Gamma,k})$, since this allows more clauses to be inferred from it using $\vdash_R$.

For example, if $\Gamma = \{(P \vee Q), (\neg P \vee Q), (P \vee \neg Q)\}$ (see Section 4.2.1) then:

$$\begin{array}{rcll} T_{\mathrm{FPE},\Gamma,0}\uparrow n & = & \emptyset & (\text{for all } n \geq 0) \\ T_{\mathrm{FPE},\Gamma,1}\uparrow n & = & \{(P),(Q)\} & (\text{for all } n \geq 1) \\ T_{\mathrm{FPE},\Gamma,k}\uparrow n & = & \{(P),(Q)\} \cup \Gamma & (\text{for all } n \geq 1 \text{ and all } k \geq 2) \end{array}$$

For another example, consider the theory $\Delta = \Gamma \cup \{(\neg P \vee \neg Q)\}$:

$$\begin{array}{rcll} T_{\mathrm{FPE},\Delta,0}\uparrow n & = & \emptyset & (\text{for all } n \geq 0) \\ T_{\mathrm{FPE},\Delta,1}\uparrow 1 & = & \{(P),(\neg P),(Q),(\neg Q)\} & \\ T_{\mathrm{FPE},\Delta,1}\uparrow 2 & = & \{\mathbf{f}\} \cup T_{\mathrm{FPE},\Delta,1}\uparrow 1 \ = \ E(\Delta,1) & \end{array}$$

The least fixpoints are given by:

$$\begin{array}{c} \mathrm{lfp}(T_{\mathrm{FPE},\Gamma,0}) = \mathrm{lfp}(T_{\mathrm{FPE},\Delta,0}) = \emptyset \\ \mathrm{lfp}(T_{\mathrm{FPE},\Gamma,1}) = \{(P),(Q)\}; \qquad \mathrm{lfp}(T_{\mathrm{FPE},\Delta,1}) = E(\Delta,1) \\ \mathrm{lfp}(T_{\mathrm{FPE},\Gamma,2}) = \Gamma \cup \{(P),(Q)\}; \qquad \mathrm{lfp}(T_{\mathrm{FPE},\Delta,2}) = E(\Delta) \end{array}$$

Note that $\mathbf{f}$ is a basic clause in $\mathrm{lfp}(T_{\mathrm{FPE},\Delta,1})$ but not in $\mathrm{lfp}(T_{\mathrm{FPE},\Delta,0})$, although it is in $E(\Delta, 0)$. This is possible, intuitively, since obtaining $\mathbf{f}$ from Δ using $\vdash_{\mathrm{FPE}}$ requires that at least one of the basic clauses in the set $\{P, Q, \neg P, \neg Q\}$ be added to Δ; this happens in $\mathrm{lfp}(T_{\mathrm{FPE},\Delta,1})$ but not in $\mathrm{lfp}(T_{\mathrm{FPE},\Delta,0})$. In general, for a theory Γ higher values of k may lead to more clauses of sizes smaller than k to be in $\mathrm{lfp}(T_{R,\Gamma,k})$ due to such "feedback" effects.

4.4.2 Properties of the Fixed-Point

We now show three significant properties of the fixpoint $\text{lfp}(T_{R,\Gamma,k})$: monotonicity, soundness, and eventual completeness. The fourth property, tractability, will be shown in Section 4.4.4. Note first that the fixpoint contains only basic clauses.

Proposition 4.12 (Monotonicity) *For any theories* Γ, Γ', *and* Δ *and any numbers* k *and* p, *if* $E(\Gamma, k) \subseteq E(\Delta, p)$ *and* $\Gamma \Leftrightarrow^*_R \Gamma' \subseteq \Delta$ *then* $\text{lfp}(T_{R,\Gamma,k}) \subseteq \text{lfp}(T_{R,\Delta,p})$.

Proof: All we need to show is that for all n, $T_{R,\Gamma,k}{\uparrow}n \subseteq T_{R,\Delta,p}{\uparrow}n$. We show this by induction on n:

($n = 0$): trivial since $T_{R,\Gamma,k}{\uparrow}0 = \emptyset$.

($n > 0$): The inductive hypothesis is that $T_{R,\Gamma,k}{\uparrow}(n-1) \subseteq T_{R,\Delta,p}{\uparrow}(n-1)$. For any basic clause μ:

$$\begin{array}{rcll}
\mu \in T_{R,\Gamma,k}{\uparrow}n & \Rightarrow & \mu \in T_{R,\Gamma,k}(T_{R,\Gamma,k}{\uparrow}(n-1)) & \text{(definition)} \\
& \Rightarrow & \Gamma \cup T_{R,\Gamma,k}{\uparrow}(n-1) \vdash_R \mu & \text{(definition)} \\
& \Rightarrow & \Delta \cup T_{R,\Gamma,k}{\uparrow}(n-1) \vdash_R \mu & \text{(modularity and Proposition 4.3)} \\
& \Rightarrow & \Delta \cup T_{R,\Delta,p}{\uparrow}(n-1) \vdash_R \mu & \text{(modularity and inductive hyp.)} \\
& \Rightarrow & \mu \in T_{R,\Delta,p}(T_{R,\Delta,p}{\uparrow}(n-1)) & \text{(defn. and } E(\Gamma,k) \subseteq E(\Delta,p)) \\
& \Rightarrow & \mu \in T_{R,\Delta,p}{\uparrow}n & \text{(definition)}
\end{array}$$

Thus, $T_{R,\Gamma,k}{\uparrow}n \subseteq T_{R,\Delta,p}{\uparrow}n$. Note the dependence on Γ' in going from Γ to Δ in the above sequence.

Thus, $\text{lfp}(T_{R,\Gamma,k}) \subseteq \text{lfp}(T_{R,\Delta,p})$. ∎

The following observations follow directly from Proposition 4.12, for any theories Γ and Δ and any numbers k and p:

1. if $k \leq p$ and $\Gamma \subseteq \Delta$ then $\text{lfp}(T_{R,\Gamma,k}) \subseteq \text{lfp}(T_{R,\Delta,p})$. This follows since $k \leq p$ and $\Gamma \subseteq \Delta$ imply that $E(\Gamma, k) \subseteq E(\Delta, p)$.
2. if $k \geq |\text{HB}(\Gamma)|$ then $\text{lfp}(T_{R,\Gamma,k}) = \text{lfp}(T_{R,\Gamma,|\text{HB}(\Gamma)|})$, since $E(\Gamma, k) = E(\Gamma) = E(\Gamma, |\text{HB}(\Gamma)|)$. Thus, the sequence of least fixpoints for increasing k's converge by the time $k = |\text{HB}(\Gamma)|$. Since we are dealing with only finite theories, this value is also finite.
3. if $\Gamma \Leftrightarrow^*_R \Delta$ and $\text{HB}(\Gamma) \subseteq \text{HB}(\Delta)$ then $\text{lfp}(T_{R,\Gamma,k}) \subseteq \text{lfp}(T_{R,\Delta,k})$. Thus, theories that have the same Herbrand base and are equivalent with respect to FPE have the same least fixpoints.

The following example shows that claim 3 above may be violated if $\text{HB}(\Gamma) \not\subseteq \text{HB}(\Delta)$. Consider $\Gamma = \{P, \neg P, Q\}$ and $\Delta = \{P, \neg P\}$. Since $\text{FPF}(\Gamma) = \{\mathbf{f}\} = \text{FPF}(\Delta)$, we have $\Gamma \Leftrightarrow^*_{\text{FPE}} \Delta$. However, $\text{HB}(\Gamma) = \{P, Q\} \not\subseteq \{P\} = \text{HB}(\Delta)$. Also, $\text{lfp}(T_{\text{FPE},\Gamma,1}) = \{(P), (\neg P), (Q), (\neg Q), \mathbf{f}\} \not\subseteq \{(P), (\neg P), \mathbf{f}\} = \text{lfp}(T_{\text{FPE},\Delta,1})$. This idea can be used to create similar examples where the two theories are satisfiable.

The next two theorems relate the basic clauses in the least fixpoint $\text{lfp}(T_{R,\Gamma,k})$ to the logical content of the theory Γ: Theorem 4.13 shows that every basic clause in the least fixpoint is logically entailed by the theory Γ; Theorem 4.14 shows that if we keep on increasing k, then eventually some fixpoint contains all the basic clauses in $E(\Gamma)$ that are entailed by Γ.

Theorem 4.13 (Soundness) *For any theory* Γ, *any basic clause* μ, *and any* k, *if* $\mu \in \text{lfp}(T_{R,\Gamma,k})$ *then* $\Gamma \models \mu$.

Proof: Directly, by the soundness of $\vdash_R$ and by induction on n, where $\text{lfp}(T_{R,\Gamma,k})$ is given by $\cup\{T_{R,\Gamma,k}{\uparrow}n \mid n \in \mathcal{N}\})$. ∎

It follows from Theorem 4.13 that the clauses added to a theory Γ for obtaining $\text{Viv}(R, \Gamma, k)$ for any number k are logically entailed by Γ. Thus, we obtain the following corollary of Theorem 4.13:

Corollary: *Any theory* Γ *is logically equivalent to the theory* $\text{Viv}(R, \Gamma, k)$ *for any number* k.

Theorem 4.14 (Eventual Completeness) *For any theory* Γ*, any* $m \geq |\text{HB}(\Gamma)|$*, and any basic clause* μ *in* $E(\Gamma)$*, if* $\Gamma \models \mu$ *then* $\mu \in \text{lfp}(T_{R,\Gamma,m})$.

Proof: (by contradiction) All we need to prove is that the theorem holds for $m = |\text{HB}(\Gamma)|$; other cases would then follow directly from Proposition 4.12. Assume now that the claim is false for $m = |\text{HB}(\Gamma)|$, i.e., there is some theory Γ, some basic clause $\mu \in E(\Gamma)$ such that $\Gamma \models \mu$ but $\mu \notin \text{lfp}(T_{R,\Gamma,m})$. For this fixed Γ, let μ be a maximal basic clause for which the theorem does not hold. Since μ is a basic clause, size of μ is at most m.

Case 1: Size of μ is m, i.e, $\text{atoms}(\mu) = \text{HB}(\Gamma) \supseteq \text{atoms}(\Gamma)$. Since $\Gamma \models \mu$, we obtain from Proposition 4.6 that $\Gamma \vdash_R \mu$, i.e., $\mu \in \text{lfp}(T_{R,\Gamma,m})$.

Case 2: Size of μ is less than m, i.e., there is an atom $p \in \text{HB}(\Gamma) - \text{atoms}(\mu)$. Thus, both $\mu \overset{\circ}{\vee} (p)$ and $\mu \overset{\circ}{\vee} (\neg p)$ are in $E(\Gamma, m)$. Since $\Gamma \models \mu$, it follows that $\Gamma \models \mu \overset{\circ}{\vee} (p)$. Since μ is a maximal clause that violates the theorem, $\mu \overset{\circ}{\vee} (p) \in \text{lfp}(T_{R,\Gamma,m})$. Similarly, $\mu \overset{\circ}{\vee} (\neg p) \in \text{lfp}(T_{R,\Gamma,m})$. From Proposition 4.7 and Proposition 4.3, we obtain that $\text{lfp}(T_{R,\Gamma,m}) \vdash_R \mu$. Since this is a fixpoint, it follows that $\mu \in \text{lfp}(T_{R,\Gamma,m})$.

Since we arrive at a contradiction in all cases, the theorem is proved. ∎

Note that eventual completeness does not follow directly from the ability of $\vdash_R$ to evaluate interpretations (Proposition 4.6). It also depends crucially on the ability of $\vdash_R$ to derive shorter clauses using case analysis on longer clauses (Proposition 4.7). The main result follows as a corollary of Theorem 4.14:

Corollary 4.15 *For any theory* Γ*,* $\text{Viv}(R, \Gamma, m)$ *is R-vivid for* $m = |\text{HB}(\Gamma)|$.

Proof: It follows from Theorem 4.14 and Proposition 4.5 that $\vdash_R$ is complete for inferring basic clauses in $E(\Gamma)$ from $\text{Viv}(R, \Gamma, m)$. Since $E(\Gamma) = E(\text{Viv}(R, \Gamma, m))$, it then follows from Proposition 4.10 that $\vdash_R$ is complete for $\text{Viv}(R, \Gamma, m)$, i.e., $\text{Viv}(R, \Gamma, m)$ is R-vivid. ∎

4.4.3 Vivid Theories and Intricacy

We have already seen in Section 4.2.1 that Horn, positive, negative, and unsatisfiable 2-CNF theories are FPE-vivid. We will now show that $\text{Viv}(\text{FPE}, \Gamma, 1)$ is FPE-vivid for any satisfiable 2-CNF theory Γ. We first give two examples and then a proposition from which the desired claim will follow. The crucial observation is that a theory $\text{Viv}(R, \Gamma, k)$ is R-vivid for some k iff each clause in $\text{lfp}(T_{R,\Gamma,i})$ for each value of i can be inferred from $\text{Viv}(R, \Gamma, k)$ using $\vdash_R$. Moreover, if Γ is unsatisfiable then $\text{Viv}(R, \Gamma, k)$ is R-vivid iff $\text{Viv}(R, \Gamma, k) \vdash_R \mathbf{f}$.

Consider the theory $\Gamma = \{(P \vee Q), (\neg P \vee Q), (P \vee \neg Q)\}$ that we saw earlier in this section. Since any clause in $\text{lfp}(T_{\text{FPE},\Gamma,k})$, for any k, is inferable from Γ using $\vdash_{\text{FPE}}$, it follows that $\Gamma = \text{Viv}(\text{FPE}, \Gamma, 0)$ is FPE-vivid. Note that $|\text{HB}(\Gamma)| = 2$.

Consider the theory $\Delta = \Gamma \cup \{(\neg P \vee \neg Q)\}$, also seen earlier in this section. Since $\mathbf{f} \in \text{lfp}(T_{\text{FPE},\Delta,1})$ but $\Delta \not\vdash_{\text{FPE}} \mathbf{f}$, $\Delta = \text{Viv}(\text{FPE}, \Delta, 0)$ is not FPE-vivid. However, $\text{Viv}(\text{FPE}, \Delta, 1)$ is FPE-vivid, since $\text{Viv}(\text{FPE}, \Delta, 1) \vdash_{\text{FPE}} \mathbf{f}$.

Proposition 4.16 $\mathbf{f} \in \text{Viv}(\text{FPE}, \Gamma, 1)$ *for any unsatisfiable 2-CNF theory* Γ.

Proof: Suppose Γ is a 2-CNF theory and $\mathbf{f} \notin \text{Viv}(\text{FPE}, \Gamma, 1)$. We will show that Γ is satisfiable by constructing a model as follows (recall that FPF is the reduction function for the rewrite system FPE):

$\Delta := FPF(\text{Viv}(\text{FPE}, \Gamma, 1))$;

```
for each p in atoms(Γ) do
    if p ∉ Δ then
        Δ := FPF(Δ ∪ {¬p})}}
```

All we need to show is that $\Delta \neq \{\mathbf{f}\}$ is an invariant maintained by the loop. We prove this by induction on the number of iterations of the loop. The invariant holds before entering the loop because $\text{Viv}(\text{FPE}, \Gamma, 1) \nvdash_{\text{FPE}} \mathbf{f}$. Since Δ is made irreducible with respect to FPE, it is the union of two sets with disjoint atoms: a set of literals, and a subset, say Δ', of the binary clauses of Γ. If a literal $\neg p$ is added to Δ in any iteration then p is not a literal in Δ and $\text{Viv}(\text{FPE}, \Gamma, 1) \nvdash_{\text{FPE}} p$. Thus, $FPF(\Delta \cup \{\neg p\}) \neq \{\mathbf{f}\}$.

Since the final Δ contains a literal for each atom and $\Delta \neq \{\mathbf{f}\}$, it follows from Proposition 4.6 that Δ is satisfiable. Hence, Γ is satisfiable. ∎

It follows from Proposition 4.16 that any $\text{Viv}(\text{FPE}, \Gamma, 1)$ is FPE-vivid for any unsatisfiable 2-CNF theory Γ. We have already seen in Section 4.2.1 that any satisfiable 2-CNF theory is FPE-vivid. Therefore, $\text{Viv}(\text{FPE}, \Gamma, 1)$ is FPE-vivid for any 2-CNF theory Γ.

Thus, there are many theories Γ for which $\text{Viv}(R, \Gamma, k)$ is R-vivid even for values of k smaller than $|\text{HB}(\Gamma)|$. Based on this observation, we define a measure on theories that indicates the difficulty of obtaining a logically equivalent R-vivid theory:

Definition 4.13 The *R-intricacy* of any theory Γ is the least value of k for which $\text{Viv}(R, \Gamma, k)$ is R-vivid. ∎

It follows that although the intricacy of any theory is at most the number of distinct atoms in the theory, it can be much lower than that for specific theories. In particular, the FPE-intricacy of any Horn, positive, negative, or satisfiable 2-CNF theory is 0, and the FPE-intricacy of any 2-CNF theory is at most 1.

4.4.4 Computing the Least Fixed-Point

We now determine the cost of computing the least fixpoint used in obtaining R-vivid theories. A straightforward way for computing the least fixpoint $\text{lfp}(T_{R,\Gamma,k})$ for any theory Γ and any number k is given in the algorithm **Compute-R-lfp** of Figure 4.1. After computing the Herbrand base and the k-extended Herbrand base, the fixpoint is built incrementally: starting with an empty set, keep adding to it the basic clauses in $E(\Gamma, k)$ that can be inferred from this set and Γ using $\vdash_R$. For any input theory Γ and any number k, it is easy to verify that **Compute-R-lfp** returns $\text{lfp}(T_{R,\Gamma,k})$. Lemma 4.17 shows the time complexity of Compute-R-lfp.

Lemma 4.17 *For any theory Γ of size n, and any numbers k and p, if each predicate in Γ has arity at most p then Compute-R-lfp(Γ,k) takes time at most $m^2 * f(k * m)$, where $m = k * (2n)^{kp+k}$ and function f provides the cost of computing $\vdash_R$.*

Proof: Consider any Γ and any fixed k and p, independent of the size of Γ. Suppose n is the size of Γ. Since the number of distinct constants and predicates in Γ is at most n, $\text{HB}(\Gamma)$ has at most n^{p+1} distinct atoms; thus, $E(\Gamma, k)$ has at most $k * (2n)^{kp+k}$ (say, m) distinct basic clauses. Since each basic clause in $E(\Gamma, k)$ has at most k literals, the size of $\text{lfp}(T_{R,\Gamma,k})$ is at most $k * m$. Since in the algorithm Δ is always a subset of $\text{lfp}(T_{R,\Gamma,k})$, we have that the size of Δ is also bounded by $k * m$.

Since each iteration of the repeat loop adds at least one new clause to Δ, there are at most m iterations. Each iteration requires testing whether $\Gamma \cup \Delta' \vdash_R \mu$ for each μ in $E(\Gamma, k) - \Delta'$, where Δ' is the current value of Δ. Each such test requires at most $f(k * m)$ time, where the function f represents the time complexity of $\vdash_R$; the total time, then, for each iteration is at most $m * f(k * m)$. Thus, the total time for all the iterations is at most $m^2 * f(k * m)$. ∎

Since the cost f of determining $\vdash_{\text{FPE}}$ is quadratic, it follows that $\text{lfp}(T_{\text{FPE},\Gamma,k})$ can be computed in time polynomial in the size of Γ but exponential in k.

Algorithm Compute-R-lfp:

```
    Input:  a theory Γ and a number k;
    Output: lfp(T_{R,Γ,k})
    compute HB(Γ) and E(Γ,k);
    Δ := ∅;
    repeat
        for each μ ∈ E(Γ,k) − Δ do
            if Γ ∪ Δ ⊢_R μ
                then Δ := Δ ∪ {μ};
    until no more changes in Δ;
    return Δ
End.
```

Figure 4.1: Algorithm to compute $\mathrm{lfp}(T_{R,\Gamma,k})$

4.5 Consequence Relations for Limited Deduction

We develop an alternate characterization of $\mathrm{Viv}(R,\Gamma,k)$ in terms of a family of increasingly complete consequence relations for limited deduction. For any admissible rewrite system R, we first extend $\vdash_R$ by adding a new inference rule to define a logically complete consequence relation $\vdash^R$. We then restrict the new inference rule to obtain the desired family $\vdash^R_k$ of increasingly complete consequence relations, where k is any natural number. We finally show that for each theory Γ and each number k: the set of clauses inferable from $\mathrm{Viv}(R,\Gamma,k)$ using $\vdash_R$ is exactly the set of clauses inferable from Γ using $\vdash^R_k$. Hence, the R-intricacy of any theory Γ is the least k for which $\vdash^R_k$ is complete.

Some other families of increasingly complete consequence relations have been previously proposed (c.f. [CS92b, CK91]). Another such family can also be directly obtained from the tractable satisfiability classes proposed in [GS88]. We will compare our family $\vdash^{\mathrm{FPE}}_k$ with these. We continue to use R to refer to any admissible rewrite system.

4.5.1 A Complete Consequence Relation

For any admissible rewrite system R, the consequence relation $\vdash_R$ is sound but may be incomplete. A source of incompleteness in $\vdash_R$ is its inability to use previously inferred clauses for inferring new clauses.

For example, for the theory $\Gamma_0 = \{(P \vee Q), (P \vee \neg Q), (\neg P \vee S)\}$, both $\Gamma_0 \vdash_{\mathrm{FPE}} P$ and $\Gamma_0 \cup \{P\} \vdash_{\mathrm{FPE}} S$, but $\Gamma_0 \not\vdash_{\mathrm{FPE}} S$. In other words, while P can be inferred from Γ_0 and S can be inferred if P is added to Γ_0, S can't be inferred from Γ_0 itself. Thus, $\vdash_{\mathrm{FPE}}$ was unable to use the previously inferred clause P to infer the new clause S.

We can extend $\vdash_R$ by adding an inference rule that provides this capability:

Definition 4.14 The consequence relation $\vdash^R$ is defined using the following two inference rules:

$$1. \quad \frac{\Gamma \vdash_R \psi}{\Gamma \vdash^R \psi}$$

$$2. \quad \frac{\Gamma \vdash^R \psi; \qquad \Gamma, \psi \vdash^R \varphi}{\Gamma \vdash^R \varphi}$$

∎

It follows from the first inference rule that $\vdash^R$ is at least as complete as $\vdash_R$. It is the second inference rule that provides the capability of using previously inferred clauses to infer new clauses. Note that Γ, ψ in the rule is the standard notation for denoting the theory $\Gamma \cup \{\psi\}$ in presenting inference rules. Inferring a clause from a theory using $\vdash^R$ requires a proper derivation, i.e., a sequence of steps, each of the form either $\Gamma \vdash_R \psi$ or $\Gamma \vdash^R \psi$; if it is of the latter form then it must have been obtained from one the above inference rules using earlier steps. It is trivial to verify that $\Gamma_0 \vdash^{\text{FPE}} S$ in the above example.

It directly follows from soundness and monotonicity of $\vdash_R$ that $\vdash^R$ is both sound and monotonic (a detailed proof can use induction on the length of derivation). It follows directly from Theorem 4.18, which is stated and proved below, and Theorem 4.14 that $\vdash^R$ is also complete.

4.5.2 A Family of Tractable Consequence Relations

The consequence relation $\vdash^R$ can be restricted to obtain consequence relations that are more complete than $\vdash_R$, but are still tractable. For instance, if ψ in Rule 2 of $\vdash^{\text{FPE}}$ is restricted to be a unit clause, then the restricted $\vdash^{\text{FPE}}$ is tractable; it is also refutation complete for 2-CNF theories, which $\vdash_{\text{FPE}}$ is not.

Thus, restricting the size of ψ in Rule 2 of $\vdash^R$ seems to be a reasonable approach for obtaining tractable consequence relations. The following inference system defines a family $\vdash^R_k$ of consequence relations, where k is any natural number:

Definition 4.15 For any natural number k, the consequence relation $\vdash^R_k$ is defined using the following two inference rules:

$$1. \quad \frac{\Gamma \vdash_R \psi}{\Gamma \vdash^R_k \psi}$$

$$2. \quad \frac{\Gamma \vdash^R_k \psi; \quad \Gamma, \psi \vdash^R_k \varphi}{\Gamma \vdash^R_k \varphi} \quad \text{for } |\psi| \leq k$$

■

The only difference from the inference rules for $\vdash^R$ is that the size of the clause ψ, which is the number of literals in ψ, in the second rule is now restricted. As for $\vdash^R$, it follows directly that for any k, $\vdash^R_k$ is sound, monotonic, and at least as complete as $\vdash_R$. In fact, it follows from Proposition 4.2 that $\vdash^R_0$ is identical to $\vdash_R$, since the only clause of size 0 is $\mathbf{f}$. It also follows directly from the inference rules that completeness of $\vdash^R_k$ is non-decreasing with increase in k: for any theory Γ, clause ψ, and number k, if $\Gamma \vdash^R_k \psi$ then $\Gamma \vdash^R_{k+1} \psi$.

Although Rule 2 explicitly allows using only one previously inferred clause, the rules of constructing proofs clearly allow for chaining: for any clauses $\psi_0, \psi_1, \ldots$ such that $|\psi_i| \leq k$ for each i, if $\Gamma \vdash^R_k \psi_n$; $\Gamma, \psi_n \vdash^R_k \psi_{n-1}; \ldots; \Gamma, \psi_n, \ldots, \psi_1 \vdash^R_k \psi_0$ then $\Gamma \vdash^R_k \psi_0$.

Theorem 4.18 *For any theory Γ, any clause ψ, and any number k:* $\text{Viv}(R, \Gamma, k) \vdash_R \psi$ *iff* $\Gamma \vdash^R_k \psi$.

Proof: It follows from Proposition 4.10 that it is sufficient to prove the claim when ψ is a basic clause in $E(\Gamma)$. Recall that $\text{Viv}(R, \Gamma, k) = \Gamma \cup \text{lfp}(T_{R,\Gamma,k})$.

(Only if) Suppose $\text{Viv}(R, \Gamma, k) \vdash_R \psi$. Since $\vdash^R_k$ is no less complete than $\vdash_R$, we have $\Gamma \cup \text{lfp}(T_{R,\Gamma,k}) \vdash^R_k \psi$. For any $\mu \in \text{lfp}(T_{R,\Gamma,k})$, it follows from the definition and finiteness of the fixpoint that $\Gamma \vdash^R_k \mu$, since $\vdash^R_k$ allows chaining. Using chaining again, we obtain that $\Gamma \vdash^R_k \psi$.

(If) Suppose $\Gamma \vdash^R_k \psi$. We show that $\text{Viv}(R, \Gamma, k) \vdash_R \psi$ by induction on the length of the derivation for $\Gamma \vdash^R_k \psi$.

In the base case, where $\Gamma \vdash_R \psi$, it follows from Proposition 4.3 that $\text{Viv}(R, \Gamma, k) \vdash_R \psi$.

For the inductive case, there is a clause φ such that $|\varphi| \leq k$, $\Gamma \vdash^R_k \varphi$, and $\Gamma, \varphi \vdash^R_k \psi$. Using the inductive assumption, we have $\text{Viv}(R, \Gamma, k) \vdash_R \varphi$ and $\text{Viv}(R, \Gamma \cup \{\varphi\}, k) \vdash_R \psi$. There are two mutually-exclusive and exhaustive cases:

1. The bag of literals in clause φ is inconsistent: it follows from Property I† that $\Gamma \Leftrightarrow^*_R \Gamma \cup \{\varphi\}$, and then from Lemma 4.11 that $\mathrm{Viv}(R, \Gamma, k) = \mathrm{Viv}(R, \Gamma \cup \{\varphi\}, k)$. Thus, $\mathrm{Viv}(R, \Gamma, k) \vdash_R \psi$.

2. Otherwise: let μ denote the clause $\vee(B \cap (\mathrm{HB}(\Gamma) \cup \sim\mathrm{HB}(\Gamma)))$, where $\varphi = \vee(B)$. It follows from Proposition 4.10 that $\mathrm{Viv}(R, \Gamma, k) \vdash_R \mu$. Thus, $\mu \in \mathrm{Viv}(R, \Gamma, k)$, since μ is a basic clause in $E(\Gamma)$. Since μ is a subclause of φ, it then follows from Property G† and $\mathrm{Viv}(R, \Gamma \cup \{\varphi\}, k) \vdash_R \psi$ that $\mathrm{Viv}(R, \Gamma, k) \vdash_R \psi$.

Thus, $\mathrm{Viv}(R, \Gamma, k) \vdash_R \psi$ in all cases. ∎

It follows that $\vdash^R_k$ is complete for Γ iff $\vdash_R$ is complete for $\mathrm{Viv}(R, \Gamma, k)$. Thus, R-intricacy of any theory Γ is the least k for which $\vdash^R_k$ is complete. Moreover, a theory is R-vivid iff $\vdash_R$ for it is identical to $\vdash^R_k$ for each k. This suggests a notion of partial R-vividness defined as follows: a theory is k-R-vivid iff $\vdash_R$ for it is identical to $\vdash^R_k$.

4.5.3 Comparison with Earlier Approaches

We compare our family $\vdash^{\text{FPE}}_k$ of tractable consequence relations with some other tractable consequence relations presented in the literature.

Relevance Logic and RP-Entailment

Belnap [Bel77] presented a 4-valued model-theory for PC, called relevance logic, whose entailment relation, say $\models_B$, is strictly weaker than $\models$, the entailment relation for classical 2-valued model theory. Intuitively, relevance logic allows equivalences based on the properties of logical operators such as commutativity, associativity, distributivity, De Morgan's laws and double negation [AB75]; for example, $\{\psi \vee \neg\neg\mu\} \models_B \mu \vee \psi$, for any formula ψ and μ. It also allows inferring clauses from their subclauses; for example, $\{(P \vee Q)\} \models_B (P \vee Q \vee R)$. However, relevance logic blocks chaining; for example, $\{P, \neg P \vee Q\} \not\models_B Q$.

Levesque [Lev84b] presented a logic of implicit and explicit beliefs, where explicit beliefs are obtained using the $\models_B$ entailment, and proved that $\Gamma \models_B \psi$ can be determined in $O(|\Gamma|\ |\psi|)$ time, if the theory Γ and the formula ψ are both in CNF. The entailment holds iff each clause in ψ is a superclause of some clause in Γ.

Frisch [Fri87] presented a 3-valued model-theory for PC, whose entailment relation, $\models_{RP}$, is strictly stronger than $\models_B$ but strictly weaker than $\models$. For the CNF case, he proved that $\Gamma \models_{RP} \psi$ can also be determined in $O(|\Gamma|\ |\psi|)$ time. He also argued that it is the strongest propositional logic that is sound but allows no chaining, and proved that $\Gamma \models_{RP} \psi$ iff $\Gamma \cup \{P \vee \neg P \mid P \text{ is an atom in } \Gamma \cup \{\psi\}\} \models_B \psi$. For example, $\models_{RP} (P \vee \neg P)$ but $\not\models_B (P \vee \neg P)$.

Since $\models_{RP} \psi$ iff ψ is a tautology iff $\neg\psi$ is unsatisfiable [Fri87], it then follows that $\models_{RP}$ is intractable, in general (specially when ψ is in disjunctive normal form). Since none of the entailment relations $\vdash^{\text{FPE}}_k$ is complete, it follows that $\models_{RP}$ in not weaker than any of them. It then follows from the relation between $\models_{RP}$ and $\models_B$ that $\models_B$ is also intractable and is not weaker than any $\vdash^{\text{FPE}}_k$ relation.

If we restrict our attention to CNF theories and formulas for which both $\models_B$ and $\models_{RP}$ are tractable, it follows from the RP-decision theorem for facts [Fri87] and the semantics of conjunction that $\Gamma \models_{RP} \psi$ iff each either clause in ψ is a superclause of some clause in Γ or ψ has complimentary literals. In either of these cases, $\Gamma \vdash_{\text{FPE}} \psi$, because of properties C, D, and E of FP. Thus, $\vdash^{\text{FPE}}_0$, which is identical to $\vdash_{\text{FPE}}$, is at least as strong as $\models_{RP}$. Since $\{P, \neg P \vee Q\} \not\models_{RP} Q$ and $\{P, \neg P \vee Q\} \vdash^{\text{FPE}}_0 Q$, it then follows that $\vdash^{\text{FPE}}_0$ is strictly stronger than both $\models_B$ and $\models_{RP}$.

Approximate Entailment

Cadoli and Schaerf [CS92a] parameterized $\models_{RP}$ by sets of propositions: their entailment relation $\models_S^3$ is defined using a 3-valued model theory which restricts each atom in the set S to the traditional 2 values.[4] Intuitively, the logic allows chaining on the atoms in the set S; for example, if $P \in S$ then $\{P, \neg P \vee Q\} \models_S^3 Q$. For the CNF case, they show that the entailment $\Gamma \models_S^3 \psi$ can also be determined in $O(|\Gamma|\ |\psi|\ 2^{|S|})$ time. For the general case, $\models_S^3$ is intractable.

For any set S of atoms, if $P \notin S$ then $\{P, \neg P \vee Q\} \not\models_S^3 Q$. Thus, $\vdash_0^{\text{FPE}}$ is not weaker than any entailment $\models_S^3$, except when S contains all atoms in the language. For any number k, let S be the set $\{P_1, \ldots, P_{k+2}\}$ and let Γ be the theory containing all $(k+2)$-clauses built from the atoms in S. It follows that $\Gamma \models_S^3 \mathbf{f}$ and $\Gamma \not\vdash_k^{\text{FPE}} \mathbf{f}$. Thus, for each number k there is a set S of size $k+2$ such that $\models_S^3$ is not weaker than $\vdash_k^{\text{FPE}}$. Thus, the two families of entailments are incomparable.

Bounded Resolution

Gallo and Scutellà [GS88] built a hierarchy, $\Gamma = \Gamma_0, \Gamma_1, \ldots$, of classes of theories in PC, such that for each Γ_k, the satisfiability problem is solvable in $O(n^{k+1})$ time, where n is the size of the theory. Büning [Bun92] defined k-resolution, a restriction on resolution that at least one parent must have at most k literals, and showed that k-resolution is refutation complete for Γ_{k-1}, but refutation-incomplete for Γ_k.

k-resolution can be used to define a family of tractable entailment relations: $\Gamma \vdash_k^B \psi$ iff $\Gamma \cup \{\neg\psi\}$ has a refutation using k-resolution (note that B here is not a rewrite system). Although the exact relation between $\vdash_k^{\text{FPE}}$ and $\vdash_k^B$ is still open, the following example shows that $\vdash_2^B$ is not stronger than $\vdash_2^{\text{FPE}}$.

Consider the theory Γ containing the following clauses:

$$
\begin{array}{ll}
(\neg P \vee \neg Q \vee S) & (\neg R \vee \neg U \vee \neg P \vee Q \vee V) \\
(\neg P \vee \neg Q \vee \neg S) & (\neg R \vee \neg U \vee \neg P \vee Q \vee \neg V) \\
(\neg R \vee U \vee P) & (\neg R \vee U \vee \neg P \vee Q \vee W) \\
(\neg R \vee \neg U \vee P) & (\neg R \vee U \vee \neg P \vee Q \vee \neg W)
\end{array}
$$

Since there is no clause in Γ with 2 literals, it follows that $\neg R$ can not be obtained from Γ using 2-resolution. Now consider the least fixpoint $\text{lfp}(T_{\text{FPE},\Gamma,2})$: Since $\Gamma \cup \{P, Q\} \Leftrightarrow_{\text{FPE}}^* \{\mathbf{f}\}$, the clause $(\neg P \vee \neg Q)$ is in the fixpoint. Since $\Gamma \cup \{(\neg P \vee \neg Q), R, U\} \Leftrightarrow_{\text{FPE}}^* \{\mathbf{f}\}$, the clause $(\neg R \vee \neg Q)$ is also in the fixpoint. Since $\Gamma \cup \{(\neg P \vee \neg Q), R, \neg U\} \Leftrightarrow_{\text{FPE}}^* \{\mathbf{f}\}$, the clause $(\neg R \vee Q)$ is also in the fixpoint. Thus, $(\neg R)$ is also in the fixpoint.

Now, consider the theory Γ' obtained from Γ by switching R and $\neg R$, and by replacing all other atoms by pairwise-distinct new atoms. Using the same argument given above, we obtain that R can not be obtained from Γ' using 2-resolution and that (R) is in the fixpoint $\text{lfp}(T_{\text{FPE},\Gamma',2})$. It then follows that $\Gamma \cup \Gamma' \vdash_2^{\text{FPE}} \mathbf{f}$, but $\Gamma \cup \Gamma' \not\vdash_2^B \mathbf{f}$.

Access-Limited Logics (ALL)

Crawford and Kuipers [CK89, CK91] presents ALL, a logic that attempts to formalize the access limitations that are inherent in a network-structured knowledge base. ALL allows retrieving only those assertions that are reachable by following an available access path. It is shown that if the access paths are bounded then reasoning is tractable. This system exhibits Socratic completeness in the sense that all facts that are logical consequence of the knowledge base can be deduced after a sequence of preliminary queries. They define a family $\vdash_{\text{ALL}}^k$ of entailment relation such that only k nesting of preliminary queries are allowed for inference in $\vdash_{\text{ALL}}^k$. Although the exact relation between $\vdash_k^{\text{FPE}}$ and $\vdash_{\text{ALL}}^k$ is still open, the following example [Cra94] shows that $\vdash_{\text{ALL}}^2$ is not stronger than $\vdash_2^{\text{FPE}}$.

Consider the theory Γ containing the following clauses:

[4] [CS92a] also defines a family of unsound but complete entailment relations, using a similar idea.

$$\begin{array}{ll} (\neg P \vee Q \vee U) & (\neg P \vee \neg Q \vee S \vee W) \\ (\neg P \vee Q \vee \neg U) & (\neg P \vee \neg Q \vee S \vee \neg W) \\ (P \vee R \vee V) & (P \vee \neg R \vee S \vee X) \\ (P \vee R \vee \neg V) & (P \vee \neg R \vee S \vee \neg X) \end{array}$$

It can be verified by an exhaustive case analysis that S can't be inferred from Γ using $\vdash^2_{\mathrm{ALL}}$. However, S is in the fixpoint $\mathrm{lfp}(T_{\mathrm{FPE},\Gamma,2})$, since the following clauses are also in the fixpoint: $(\neg P \vee Q)$, $(P \vee R)$, $(\neg P \vee S)$, and $(P \vee S)$.

4.6 Conclusions

We used the rewrite system FPE to define an incomplete but efficient consequence relation $\vdash_{\mathrm{FPE}}$ for inferring clauses from arbitrary theories in PCE: $\Gamma \vdash_{\mathrm{FPE}} \psi$ iff $\Gamma \cup [\![\sim\psi]\!] \Rightarrow^*_{\mathrm{FPE}} [\![\mathbf{f}]\!]$. We proved that $\vdash_{\mathrm{FPE}}$ is complete for (inferring clauses from) Horn, positive, negative, and satisfiable 2-CNF theories. Theories for which $\vdash_{\mathrm{FPE}}$ is complete are called vivid theories. We also proved that $\vdash_{\mathrm{FPE}}$ is complete for inferring clauses from a theory iff it is complete for inferring its basic clauses that do not contain logical constants and repetition of atoms and are constructed from the predicates and constants appearing in the theory.

We then used $\vdash_{\mathrm{FPE}}$ to define a function Viv such that for every Γ there is a k for which $\mathrm{Viv}(FPE, \Gamma, k)$ is FPE-vivid; the least such k is called the FPE-intricacy of Γ. $\mathrm{Viv}(FPE, \Gamma, k)$ augments the theory Γ by the least fixpoint of the operator $T_{\mathrm{FPE},\Gamma,k}$ which is defined as $T_{\mathrm{FPE},\Gamma,k}(S) = \{\mu \in E(\Gamma, k) \mid \Gamma \cup S \vdash_{\mathrm{FPE}} \mu\}$, where S is any set of k-clauses, which are basic clauses of Γ with at most k literals. The least fixpoint $\mathrm{lfp}(T_{R,\Gamma,k})$, also called the kth fixpoint of Γ, can be obtained in time polynomial in the size of Γ, but exponential in k. Thus, it might be computationally advantageous to pre-process theories with low FPE-intricacy by making them FPE-vivid, so that the relatively efficient $\vdash_{\mathrm{FPE}}$ is sound and complete for deriving clauses from them. Since $\mathrm{Viv}(\mathrm{FPE}, \Gamma, 0) = \Gamma$, the FPE-intricacy of any Horn, positive, negative, or satisfiable 2-CNF theory is 0. We also proved that the FPE-intricacy of any unsatisfiable 2-CNF theory is at most 1.

We also presented an alternate characterization of $\mathrm{Viv}(FPE, \Gamma, k)$ in terms of a family $\vdash^{\mathrm{FPE}}_k$ of increasingly complete consequence relations for limited inference: the set of clauses inferable from $\mathrm{Viv}(FPE, \Gamma, k)$ using $\vdash_{\mathrm{FPE}}$ is exactly the set of clauses inferable from Γ using $\vdash^{\mathrm{FPE}}_k$. For any number k, $\vdash^{\mathrm{FPE}}_k$ extends $\vdash_{\mathrm{FPE}}$ by allowing chaining on clauses of size at most k: if there is a clause ψ with at most k literals such that $\Gamma \vdash^{\mathrm{FPE}}_k \psi$ and $\Gamma \cup \{\psi\} \vdash^{\mathrm{FPE}}_k \varphi$ then $\Gamma \vdash^{\mathrm{FPE}}_k \varphi$. Note that ψ here is not required to be a basic clause. Since $\mathrm{Viv}(FPE, \Gamma, k)$ is FPE-vivid for some k, the family $\vdash^{\mathrm{FPE}}_k$ converges to a complete consequence relation for every theory.

Our results did not depend on the exact details of FPE; they hold for any admissible rewrite system, a notion developed by abstracting out some high-level properties of FPE.

Chapter 5

Tractable Cases of Reasoning

5.1 Overview

In this chapter, we pursue the first approach to the intractability of reasoning, i.e., identifying tractable cases (see Section 1.1). The results of the previous chapter show that clauses can be inferred from a theory in time polynomial in its size but exponential in its intricacy[1]. Thus, for any number k, if there is a class of theories each of whose intricacy is less than k, then the problem of inferring clauses from these theories is tractable. Suppose we are interested in the (more restricted) question of determining satisfiability of theories. Although satisfiability is of course tractable for a class of theories that all have intricacy at most k, for some fixed constant k, it is enough for tractability if all the *unsatisfiable* theories in the class have intricacy at most k. Although this "bounded intricacy" criterion is a sufficient condition for tractability, we show that there are tractable classes that do not have bounded intricacy. We then show that some tractable classes already proposed in the literature have bounded intricacy. We also describe some new tractable classes using the bounded intricacy criterion. Although bounding the intricacy of satisfiable theories also produces tractable classes, we have not been able to use this result to describe any non-trivial tractable class that can not be described by the bounded intricacy criterion.

The tractable classes we investigate arise from reasoning problems in the areas of constraint satisfaction, disjunctive databases, and disjunctive logic programs. The problems we consider are all polynomially reducible to the (un)satisfiability problem for PCE — each instance of the problem is translated to a theory in PCE, whose satisfiability provides the answer for that instance. For constraint satisfaction problems, we show that the induced width [DP88] of any inconsistent network is always greater than the intricacy of its translated theory (within a difference of 2), and that the intricacy of the translation of any network with only functional constraints [DH91] is 1. We then present a new family of tractable networks based on bounded intricacy that combines the intuitions behind bounded induced-width and functional constraints. For disjunctive databases, we show that the translations of each tractable class of querying identified in [IMV94] have bounded intricacy. Our proofs of these results rely crucially on the results in [DP88, IMV94]. For disjunctive logic programming [LJR92], we identify a new family of tractable programs based on bounded intricacy criterion.

We show bounded intricacy for a class C of theories by proving that there is a number k such that $\mathbf{f}$ is in $\mathrm{lfp}(T_{R,\Gamma,k})$ for each unsatisfiable theory Γ in C. A theory Γ for which $\mathbf{f} \notin \mathrm{lfp}(T_{R,\Gamma,k})$ will be called a k-consistent theory. Thus, the intricacy of an unsatisfiable theory is the least k for which it is not k-consistent. Our usual technique for proving the k-inconsistency of a theory will be to show that there is a clausal subtheory of the theory such that each clause in the conjunctive normal form of the complement of the subtheory is in the least fixpoint whose index (see Section 4.4.1) is the number of clauses in the subtheory.

We continue to use PCE, using its alternative syntax as presented in Section 2.3.1, except for Section 5.5,

[1] Since we will restrict to the particular rewrite system FPE in this chapter, we will drop the prefix "FPE-" from "FPE-intricacy" and "FPE-vivid".

in which we also use a larger fragment of first-order logic that includes quantifiers. However, we use the specific rewrite system FPE presented in Section 2.8 for the notions of fixpoints, vividness, and intricacy. Since we do not require the equality predicate $\doteq$ in the sections on constraint satisfaction problems and disjunctive logic programming, and FPE is identical to FP for PCE without equality, we use FP instead of FPE in those sections.

5.2 Tractable Satisfiability Classes

We define the satisfiability problem for a class of theories and present two independent criteria based on intricacy that guarantee its tractability. We prove that these criteria are not necessary for tractability by presenting a tractable class that violates both the criteria.

Definition 5.1 The *satisfiability problem* for a class S of theories in PCE is the following decision problem:

Input:	any theory $\Gamma \in S$;
Output:	"yes" iff Γ is satisfiable.

Satisfiability is *tractable* for S iff the satisfiability problem for S is in PTIME. ■

Theorem 5.1 below presents a criterion for tractable satisfiability: there is a number k such that all the unsatisfiable theories in the class have intricacy at most k. The basic idea is that the kth least fixpoint will contain $\mathbf{f}$ for exactly all the unsatisfiable theories in S. We will refer to this as the bounded intricacy criterion — a class of theories is said to have *bounded intricacy* if there is a number k such that each unsatisfiable theory in the class has intricacy $\leq k$.

Theorem 5.1 (Unsatisfiable intricacy) *For any class S of theories and any number k, if the intricacy of each unsatisfiable theory in S is at most k then satisfiability is tractable for S.*

Proof: Consider the algorithm **Inc-sat**, which, given a theory Γ in S and a number k as input, first computes the fixpoint $\mathrm{lfp}(T_{R,\Gamma,k})$ by calling **Compute-FPE-lfp**, and then returns "no" if the fixpoint contains the clause $\mathbf{f}$, and "yes" otherwise. We show that **Inc-sat** is both tractable and correct.

Since the time complexity of **Inc-sat** is dominated by the call to **Compute-FPE-lfp**, it follows from Lemma 4.17 that **Inc-sat** runs in time polynomial in the size of Γ (but exponential in k). Since k is fixed for S, **Inc-sat** is in PTIME for S, i.e., tractable.

For correctness, if $\mathbf{f} \in \mathrm{lfp}(T_{\Gamma,k})$ then it follows from Theorem 4.13 that Γ is unsatisfiable. For the other direction, if Γ is unsatisfiable then $\mathrm{Viv}(\Gamma, k)$ is vivid, since the intricacy of Γ is at most k. It then follows from $\Gamma \models \mathbf{f}$ that $\mathrm{Viv}(\Gamma, k) \vdash_{\text{FPE}} \mathbf{f}$, and then from the fixpoint construction that $\mathbf{f} \in \mathrm{lfp}(T_{\Gamma,k})$. Thus, **Inc-sat** returns "yes" for a theory in S iff it is satisfiable. ■

Theorem 5.2 below presents the counterpart of bounded intricacy criterion for the satisfiable theories: the existence of a number k such that all the satisfiable theories in the class have intricacy at most k. The basic idea is to make each satisfiable theory vivid by adding to it the kth least fixpoint. Satisfiability can then be tested by constructing a model of the vivid theory — this model is constructed by adding one literal at a time and checking for inconsistency (using FPE) at each step. This process of model building succeeds without leading to any dead ends exactly for the satisfiable theories in S.

Theorem 5.2 (Satisfiable intricacy) *For any set S of theories and any number k, if the intricacy of each satisfiable theory in S is at most k then satisfiability is tractable for S.*

Proof: Consider the algorithm **Cons-sat** of Figure 5.1. It can be easily verified using Theorem 3.30 and Lemma 4.17 that **Cons-sat** has time complexity polynomial in the size of Γ (but exponential in k). Since k is fixed for S, **Cons-sat** is in PTIME for S, i.e., tractable. All we need to show is that it is correct.

Algorithm Cons-sat:

Input: any theory Γ and a number k s.t.
if Γ is satisfiable
then the intricacy of Γ is at most k;

Output: ''yes'' iff Γ is satisfiable,
''no'' otherwise.

$\Delta := \Gamma \cup$ Compute-FPE-lfp(Γ,k); $\mu := \mathbf{f}$;
while (atoms(μ) $\neq$ atoms(Γ)) do {
  select any α in atoms(Γ) $-$ atoms(μ);
  if $\Delta \vdash_{\text{FPE}} \mu \overset{\circ}{\vee} \alpha$
    then $\mu := \mu \overset{\circ}{\vee} \neg\alpha$
    else $\mu := \mu \overset{\circ}{\vee} \alpha$;
}
if $\Delta \vdash_{\text{FPE}} \mu$
  then return ''no''
  else return ''yes'';

end.

Figure 5.1: An Algorithm to determine satisfiability

Let μ_i be the value of μ after the ith iteration of the while loop, and let m be the number of atoms in atoms(Γ). It follows from the corollary of Theorem 4.13 that $\Gamma \equiv \Delta$. Note that atoms(Γ) = atoms(μ_m), because each of the m iterations of the loop adds a new atom from atoms(Γ) to μ.

Suppose **Cons-sat** returns "yes" for a theory Γ. Thus, $\Delta \not\vdash_{\text{FPE}} \mu_m$ from which we obtain using Propositions 4.3 and 4.6 that $\Delta \not\models \mu_m$, i.e., Δ is satisfiable. It follows from the corollary of Theorem 4.13 that Γ is also satisfiable.

Now suppose that Γ is satisfiable, i.e., Δ is vivid. We can show by induction on i that $\Delta \not\models \mu_i$ for all i. The base case, when $i = 0$, is trivial since $\mu_0 = \mathbf{f}$ and Δ is satisfiable.There are two cases to consider for the inductive case, when $i > 0$ and the inductive assumption is that $\Delta \not\models \mu_{i-1}$:

$\Delta \vdash_{\text{FPE}} \mu_{i-1} \overset{\circ}{\vee} \alpha$: it follows that $\Delta \not\models \mu_{i-1} \overset{\circ}{\vee} \neg\alpha = \mu_i$; otherwise $\Delta \models \mu_{i-1}$, a contradiction;

otherwise: $\Delta \not\vdash_{\text{FPE}} \mu_{i-1} \overset{\circ}{\vee} \alpha = \mu_i$. Since Δ is vivid, we then have $\Delta \not\models \mu_i$.

Thus, $\Delta \not\vdash_{\text{FPE}} \mu_m$, from which it follows that **Cons-sat** returns "yes". ■

The criteria in Theorems 5.1 and 5.2 are independent, since classes that satisfy any one criterion may not satisfy the other. For example, consider the class C_1 containing all satisfiable theories in PCE and all unsatisfiable Horn theories, and the class C_2 containing all unsatisfiable theories in PCE and all satisfiable Horn theories. Since the intricacy of any Horn theory is 0, class C_1 satisfies the criterion of Theorem 5.1 only (i.e., C_1 has bounded intricacy but does not satisfy the criterion of Theorem 5.2), and class C_2 satisfies the criterion of Theorem 5.2 only. In later sections, we will present some less trivial classes with bounded intricacy. However, we have not yet found any non-trivial classes that satisfy the criterion of Theorem 5.2 alone.

The criteria given in Theorems 5.1 and 5.2 are only sufficient conditions for tractability. We now show that these criteria are not necessary for tractability by presenting a tractable class, call it C_3, of theories which violate both the criteria.

Class C_3 is obtained by encoding the pigeon-hole principle (c.f. [Coo76, CS88]), according to which it is not possible to assign $n+1$ pigeons to n holes (for any number n) such that no two pigeons are in the same hole. For each n, we construct a pigeon-hole theory $P(n)$ that captures the relevant constraints and show that the intricacy of $P(n)$ is at least $n-1$.

Let $x{:}y$ be an atom that is *true* iff pigeon x is assigned to hole y. The pigeon-hole theory $P(n)$ for any number n is defined as follows:

$$\begin{aligned} P(n) \quad = \quad & \{(i{:}1 \vee \ldots \vee i{:}n) \mid i \in 1, \ldots, (n+1)\} \cup \\ & \{(\neg i{:}k \vee \neg j{:}k) \mid i, j \in 1, \ldots, (n+1); i \neq j; k \in 1, \ldots, n\} \end{aligned}$$

The positive formulas in $P(n)$ ensure that each pigeon is assigned to at least one hole, while the negative formulas ensure that no two pigeons are assigned to the same hole. For example, $P(0) = \{\mathbf{f}\}$, $P(1) = \{(1{:}1), (2{:}1), (\neg 1{:}1 \vee \neg 2{:}1)\}$, and $P(2)$ contains the following formulas:

$$\begin{array}{lll} (1{:}1 \vee 1{:}2) & (2{:}1 \vee 2{:}2) & (3{:}1 \vee 3{:}2) \\ (\neg 1{:}1 \vee \neg 2{:}1) & (\neg 1{:}1 \vee \neg 3{:}1) & (\neg 2{:}1 \vee \neg 3{:}1) \\ (\neg 1{:}2 \vee \neg 2{:}2) & (\neg 1{:}2 \vee \neg 3{:}1) & (\neg 2{:}2 \vee \neg 3{:}2) \end{array}$$

The class C_3 contains all satisfiable theories in PCE and all pigeon-hole theories $P(n)$. Since each $P(n)$ is unsatisfiable and can be easily detected by its syntactic structure, C_3 is a tractable class. Since there is no number k which bounds the intricacy of all satisfiable theories in C_3, the criterion of Theorem 5.2 is violated. We will now show that C_3 does not have bounded intricacy.

Proposition 5.3 *For any number n and any clause $\mu \in E(P(n))$, if $P(n) \vdash_{\text{FPE}} \mu$ then either μ has a subclause in $P(n)$ or $|\mu| \geq (n-1)$.*

Proof: We prove this by induction on n. For $n < 2$, the claim is trivial since the size of each clause in $E(P(n))$ is at least $(n-1)$. The claim is trivial also for $n = 2$ since the only clause in $E(P(n))$ of size smaller than $(n-1)$ is $\mathbf{f}$ and $P(2) \not\vdash_{\text{FPE}} \mathbf{f}$. For the inductive case, we only have to consider $n \geq 3$. There are two cases:

μ is a positive clause: it follows from Property C† that $\sim\mu$ reduces to a set, say A, of negative literals. After propagating the literals in A through the binary clauses, some may disappear (by becoming true), but none generate new literals. Thus the only way $\mathbf{f}$ can be obtained is because of substitution in one of the positive clauses, with n literals. If fewer than $n-1$ variables are substituted in all of these clauses, the result is a theory with only binary or higher clauses, which FPE cannot simplify to $\mathbf{f}$. Therefore, in some positive clause $n-1$ or more literals are substituted, i.e., $|\mu| \geq (n-1)$.

Otherwise: Without any loss of generality, assume that μ has the negative literal $\neg(n+1){:}n$, i.e., there is a clause $\sigma \in E(P(n-1))$ such that $\mu = \sigma \overset{\circ}{\vee} (\neg(n+1){:}n)$.

$$\begin{aligned} P(n) \cup \{\sim\mu\} \quad & \Leftrightarrow^*_{\text{FPE}} \quad P(n) \cup \{\sim\sigma\} \cup \{(n+1){:}n\} \quad \text{(Property C†)} \\ & \Leftrightarrow^*_{\text{FPE}} \quad P(n-1) \cup \{\sim\sigma\} \cup \{(n+1){:}n\} \cup \{\neg i{:}n \mid i \in 1, \ldots, n\} \\ & \qquad\quad \cup \{\neg(n+1){:}k \mid k \in 1, \ldots, (n-1)\} \\ & \qquad\quad \text{(Properties C†, D†, and E†)} \end{aligned}$$

There are several subcases:

Case (a): $(n+1){:}n$ is a literal in σ — impossible, since μ is a basic clause.

Case (b): for some k, $\neg(n+1){:}k$ is a literal in σ — $(\neg(n+1){:}k \vee \neg(n+1){:}n)$ in $P(n)$ is a subclause of μ.

Case (c): for some i, $\neg i{:}n$ is a literal in σ — $(\neg i{:}n \vee \neg(n+1){:}n)$ in $P(n)$ is a subclause of μ.

Otherwise: Since $\text{atoms}(P(n-1) \cup \{\sim\sigma\} \cup \{(n+1){:}n\}) \cap \text{atoms}(\{\neg i{:}n \mid i \in 1,\ldots,n\} \cup \{\neg(n+1){:}k \mid k \in 1,\ldots,(n-1)\}) = \emptyset$, it follows from Property H† that $P(n) \cup \{\sim\mu\} \Leftrightarrow^*_{\text{FPE}} \{\mathbf{f}\}$ iff $P(n-1) \cup \{\sim\sigma\} \Leftrightarrow^*_{\text{FPE}} \{\mathbf{f}\}$. Thus, if $P(n) \vdash_{\text{FPE}} \mu$ then $P(n-1) \vdash_{\text{FPE}} \sigma$. The claim then follows from the inductive assumption.

Thus, the claim follows in all cases. ∎

It follows from the above proposition that all the basic clauses in the $(n-2)$nd fixpoint of $P(n)$ (where $n \geq 2$) are superclauses of those in $P(n)$. Thus, $\text{Viv}(P(n), n-2)$ is not vivid, since $P(n) \models \mathbf{f}$ but $\text{Viv}(P(n), n-2) \not\vdash_{\text{FPE}} \mathbf{f}$. It follows that the intricacy of $P(n)$ is at least $(n-1)$. Thus, the class C_3 does not have bounded intricacy.

Note that the exact details of the satisfiable theories in the class C_3 is not important — the only requirement is that the condition of Theorem 5.2 is violated, i.e., there is no number k which bounds the intricacy of all satisfiable theories in C_3.

McCarty [McC95] observed that theories in class C_3 can not be enumerated easily, since it contains all satisfiable theories and no unsatisfiable theory other than pigeon-hole theories. He suggested two other classes that are "better" examples for showing that the criteria of Theorems 5.1 and 5.2 are not necessary for tractability:

1. Consider the class C_4 of all theories in PCE, each of which is syntactically tagged correctly as satisfiable or unsatisfiable. The tag may be as simple as presence or absence of a particular atom (new) as the first clause. The tractable class C_4 violates the criteria of Theorems 5.1 and 5.2.
2. For each pigeon-hole theory $P(n)$, consider the theory $P'(n)$ which restricts the number of pigeons to n (instead of $n+1$). Consider the class C_5 of all theories $P(n)$ and $P'(n)$. The class C_5 is tractable and violates the criteria of Theorem 5.1. It is still open whether C_5 violates the criteria of Theorem 5.2.

The class C_4 is almost as difficult to enumerate as the class C_3, but is conceptually much simpler and does not require a long proof. On the other hand, the class C_5 is very easy to enumerate but requires an additional proof.

5.3 Tools for Proving Bounded Intricacy

We present some technical results that will be useful in later sections for proving that a class of theories has bounded intricacy.

The bounded intricacy criterion requires the existence of a number k such that each unsatisfiable theory in the given class, say C, has intricacy at most k. This is identical to requiring $\text{Viv}(\Gamma, k)$ to be vivid for each unsatisfiable theory Γ in C, which in turn is identical to requiring that $\mathbf{f}$ be in $\text{lfp}(T_{R,\Gamma,k})$ for each unsatisfiable theory Γ in C. Since we will use this condition very often, we introduce a convenient notation to express this:

Definition 5.2 For any number k, a theory Γ is *k-consistent* iff $\mathbf{f} \notin \text{lfp}(T_{\Gamma,k})$. ∎

Thus, our technique for proving bounded intricacy for a class C will be to show that there is a number k such that no unsatisfiable theory in C is k-consistent. For this, we will show that the least fixpoint contains certain clauses which require that $\mathbf{f}$ also be in the least fixpoint. We first define these product clauses, and then prove the desired claim in Proposition 5.4.

Definition 5.3 For any basic clausal theory $\Gamma = [\![\mu_1, \ldots, \mu_n]\!]$ where $n \geq 1$, and $\mu_1, \ldots, \mu_n$ are compatible:

$$\text{product}(\Gamma) = \left[\!\!\left[(\sim\alpha_1 \overset{\circ}{\vee} \ldots \overset{\circ}{\vee} \sim\alpha_n) \mid \text{ fact } \alpha_i \text{ is a subformula of } \mu_i \text{ for each } i \in 1\ldots n\right]\!\!\right]$$

If $\Gamma = [\![\,]\!]$ then $\text{product}(\Gamma)$ is defined to be the theory $[\![\mathbf{f}]\!]$. ∎

For example, if $\Gamma = [\![(a \doteq a_1 \vee a \doteq a_2 \vee a \doteq a_3), (b \doteq b_1 \vee b \doteq b_2)]\!]$ then product(Γ) is the theory given below:

$$\begin{array}{lll}[\![a \not\doteq a_1 \vee b \not\doteq b_1, & a \not\doteq a_2 \vee b \not\doteq b_1, & a \not\doteq a_3 \vee b \not\doteq b_1, \\ \;\; a \not\doteq a_1 \vee b \not\doteq b_2, & a \not\doteq a_2 \vee b \not\doteq b_2, & a \not\doteq a_3 \vee b \not\doteq b_2]\!]\end{array}$$

As a special case, if $\mathbf{f} \in \Gamma$ then product(Γ) = $[\![]\!]$. Since product(Γ) is just the clausal form of the negation of Γ, obtained by applying de Morgan's laws and distributivity, it follows that $\Gamma \cup \text{product}(\Gamma)$ is unsatisfiable for any clausal theory Γ. The next proposition shows that if the product of some basic clauses in a theory is contained in any least fixpoint of the theory, then $\mathbf{f}$ must be in that least fixpoint. Note that the product is always a basic clausal theory, since μ_is are compatible and the use of "merger" (instead of regular disjunction, defined in Section 4.2.2) in the definition removes duplicate literals in the clauses. For example,

$$\text{product}([\![(a \vee b), (b \vee c)]\!]) = [\![(\neg a \vee \neg b), (\neg a \vee \neg c), (\neg b)]\!]$$

Proposition 5.4 *For any theories Γ and Π and any number k, if $\Pi \subseteq \Gamma \cap E(\Gamma)$, the clauses in Π are compatible, and* product(Π) $\subseteq$ lfp($T_{\Gamma,k}$)*, then Γ is not k-consistent.*

Proof: Suppose $\Pi \subseteq \Gamma \cap E(\Gamma)$ and product(Π) $\subseteq$ lfp($T_{\Gamma,k}$).

Case I ($\Pi = \emptyset$):

$$\begin{array}{rcll}\Pi = \emptyset & \Rightarrow & \text{product}(\Pi) = \{\mathbf{f}\} & \text{(definition)} \\ & \Rightarrow & \mathbf{f} \in \text{lfp}(T_{\Gamma,k}) & (\text{product}(\Pi) \subseteq \text{lfp}(T_{\Gamma,k})) \\ & \Rightarrow & \Gamma \text{ is not } k\text{-consistent} & \text{(definition)}\end{array}$$

Case II ($\mathbf{f} \in \Pi$): It follows from Propositions 4.5 and 4.12 that $\mathbf{f} \in$ lfp($T_{\Gamma,k}$) for each k.

Case III (otherwise): Suppose Π has p clauses, where $p > 0$. Since $\mathbf{f} \notin \Pi$, product(Π) is not empty. Also, since each clause in product(Π) has size p and product(Π) $\subseteq$ lfp($T_{\Gamma,k}$), it follows that $p \leq k$. Let Π' be the set of all subclauses of clauses in product(Π). We claim that $\Pi' \subseteq$ lfp($T_{\Gamma,k}$). If this claim is true then $\mathbf{f} \in$ lfp($T_{\Gamma,k}$) since $\mathbf{f} \in \Pi'$; thus, Γ is not k-consistent (as argued in Case II above). Thus, we only need to prove this claim.

Assume the claim is false, i.e., $\Pi' -$ lfp($T_{\Gamma,k}$) is not empty. Let π be a maximal clause in $\Pi' -$ lfp($T_{\Gamma,k}$), i.e., there is no clause π' in Π' such that π is a proper subclause of π' and $\pi' \notin$ lfp($T_{\Gamma,k}$).

Since product(Π) $\subseteq$ lfp($T_{\Gamma,k}$), $\pi \notin$ product(Π). Hence, π is a proper subclause of some clause in product(Π), so size of π is less than p. Thus, there is at least one clause $\mu = (\alpha_1 \vee \ldots \vee \alpha_n) \in \Pi$ which does not contribute negated literals to π; moreover, $n > 0$ and atoms(π) $\cap$ atoms(μ) $= \emptyset$. Thus, for each $i \in 1 \ldots n$, $\pi \overset{\circ}{\vee} \sim \alpha_i$ is a basic clause in Π' which belongs to lfp($T_{\Gamma,k}$), since π was the maximal clause not belonging to it. It follows that

$$\{(\alpha_1 \vee \ldots \vee \alpha_n)\} \cup \{\pi \overset{\circ}{\vee} \sim \alpha_i \mid i \in 1 \ldots n\} \subseteq \Gamma \cup \text{lfp}(T_{\Gamma,k})$$

Thus, $\Gamma \cup$ lfp($T_{\Gamma,k}$) $\vdash_{\text{FPE}} \pi$ using Property C† and Proposition 4.8. From the definition of lfp($T_{\Gamma,k}$), we then have $\pi \in$ lfp($T_{\Gamma,k}$), which is a contradiction. Thus the claim is true.

Thus, in all cases Γ is not k-consistent. ■

Therefore, in order to show that a theory Γ is not k-consistent, it is sufficient to find a compatible clausal subtheory Π of Γ such that product(Π) $\subseteq$ lfp($T_{\Gamma,k}$). In order to show that a basic clause is in a least fixpoint, we will often use the following proposition:

Proposition 5.5 *For any theory Γ, any numbers k and p, and any basic clause $\mu \in E(\Gamma, p)$, if $\Gamma \cup \{\sim\mu\}$ is not k-consistent then $\mu \in$ lfp($T_{\Gamma,k+p}$).*

Proof: Since μ is compatible with $\mathbf{f}$:

$$\begin{array}{lll} \Gamma \cup \{\sim\mu\} \text{ is not } k\text{-consistent} & \Rightarrow \mathbf{f} \in \mathrm{lfp}(T_{\Gamma\cup\{\sim\mu\},k}) & \text{(definition)} \\ & \Rightarrow \mathbf{f} \mathbin{\mathring{\vee}} \mu \in \mathrm{lfp}(T_{\Gamma,k+p}) & \text{(Lemma 5.6, proven below)} \\ & \Rightarrow \mu \in \mathrm{lfp}(T_{\Gamma,k+p}) & (\mu = \mathbf{f} \mathbin{\mathring{\vee}} \mu) \end{array}$$

∎

Lemma 5.6 *For any theory Γ, compatible basic clauses σ and μ, and any numbers k and p, if $\mu \in E(\Gamma, p)$ and $\sigma \in \mathrm{lfp}(T_{\Gamma\cup\{\sim\mu\},k})$ then $\sigma \mathbin{\mathring{\vee}} \mu \in \mathrm{lfp}(T_{\Gamma,k+p})$.*

Proof: Let $\Delta = \Gamma \cup \{\sim\mu\}$. By definition

$$\begin{array}{rcl} \mathrm{lfp}(T_{\Gamma,k+p}) & = & \cup\{T_{\Gamma,k+p}{\uparrow}n \mid n \in \mathcal{N}\} \\ \mathrm{lfp}(T_{\Delta,k}) & = & \cup\{T_{\Delta,k}{\uparrow}n \mid n \in \mathcal{N}\} \end{array}$$

We will prove (by induction on n) that for every n and every basic clause σ compatible with μ, if $\sigma \in T_{\Delta,k}{\uparrow}n$ then $\sigma \mathbin{\mathring{\vee}} \mu \in T_{\Gamma,k+p}{\uparrow}n$.

($n = 0$) trivial since $T_{\Delta,k}{\uparrow}0 = \emptyset$.

($n > 0$) Since $\sigma \in T_{\Delta,k}$, it must be the case that $\sigma \in E(\Delta, k)$. It follows that $\sigma \mathbin{\mathring{\vee}} \mu \in E(\Gamma, k+p)$, since $\mu \in E(\Gamma, p)$ and $\Delta = \Gamma \cup \{\sim\mu\}$.

The inductive hypothesis is that for every basic clause σ compatible with μ, if $\sigma \in T_{\Delta,k}{\uparrow}(n-1)$ then $\sigma \mathbin{\mathring{\vee}} \mu \in T_{\Gamma,k+p}{\uparrow}(n-1)$. It follows that $T_{\Delta,k}{\uparrow}(n-1) \mathbin{\mathring{\vee}} \{\mu\} \subseteq T_{\Gamma,k+p}{\uparrow}(n-1)$.

$$\begin{array}{lll} \sigma \in T_{\Delta,k}{\uparrow}n & \Rightarrow \Delta \cup T_{\Delta,k}{\uparrow}(n-1) \vdash_{\text{FPE}} \sigma & \text{(definition)} \\ & \Rightarrow \Gamma \cup \{\sim\sigma, \sim\mu\} \cup T_{\Delta,k}{\uparrow}(n-1) \Leftrightarrow^*_{\text{FPE}} \{\mathbf{f}\} & \text{(definition)} \\ & \Rightarrow \Gamma \cup (T_{\Delta,k}{\uparrow}(n-1) \mathbin{\mathring{\vee}} \{\mu\}) \cup \{\sim(\sigma \mathbin{\mathring{\vee}} \mu)\} \Leftrightarrow^*_{\text{FPE}} \{\mathbf{f}\} & \\ & \quad \text{(Proposition 4.9 and modularity)} & \\ & \Rightarrow \Gamma \cup (T_{\Delta,k}{\uparrow}(n-1) \mathbin{\mathring{\vee}} \{\mu\}) \vdash_{\text{FPE}} \sigma \mathbin{\mathring{\vee}} \mu & \text{(definition)} \\ & \Rightarrow \Gamma \cup T_{\Gamma,k+p}{\uparrow}(n-1) \vdash_{\text{FPE}} \sigma \mathbin{\mathring{\vee}} \mu & \\ & \quad \text{(inductive hypothesis and Proposition 4.3)} & \\ & \Rightarrow \sigma \mathbin{\mathring{\vee}} \mu \in T_{\Gamma,k+p}{\uparrow}n & \text{(definition)} \end{array}$$

This proves the lemma. ∎

In the next three sections, we will use the above results in proving bounded intricacy for some reasoning problems in the areas of constraint satisfaction, disjunctive databases, and disjunctive logic programs.

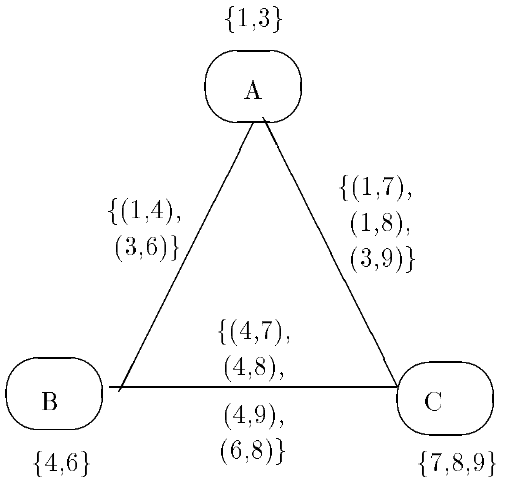

Figure 5.2: A constraint network

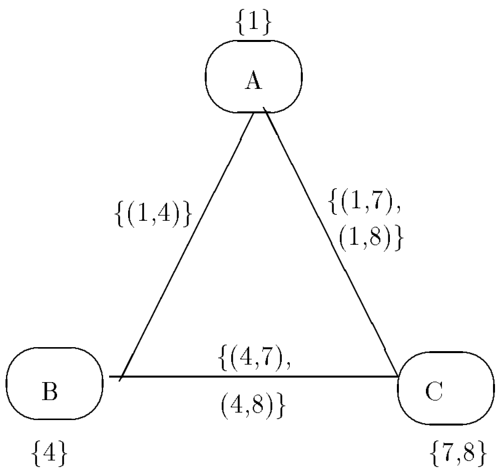

Figure 5.3: The simplified network

5.4 Constraint Satisfaction Problems

5.4.1 Overview

A constraint satisfaction problem (CSP) [Mac77, Fre78] is specified by a finite set of variables and a set of constraints on subsets of these variables limiting the values they can take. For example, a constraint might restrict the value of a variable to be greater than that of another variable. Recall from Chapter 1 that a large number of problems in AI and other areas of computer science can be viewed as special cases of CSP. A CSP is said to be consistent iff there is an assignment of a value to each variable such that all the constraints are satisfied. Determining consistency is known to be intractable [Fre78] even for *constraint networks*, a restricted class of CSPs in which all the constraints are explicitly provided as sets of tuples. Identifying classes of constraint networks for which consistency is provably tractable has generated considerable interest (c.f. [Fre82, DP87, DH91]).

In this section, we present a quadratic-time translation of constraint networks to a restricted class of theories in PC (i.e., PCE without equality), called constraint theories. Apart from the formulas encoding the constraints, these theories have formulas that encode the facts that each variable should be assigned exactly one value in its specified domain. It follows from the results of previous sections that the consistency problem is tractable for classes of constraint theories with bounded intricacy. We show that some tractable classes of constraint networks previously identified, for example, in [Fre82, DP87, DH91], translate to classes of constraint theories with bounded intricacy. We also use this criterion to describe a new, more inclusive tractable class of constraint networks. Since the rewrite system FPE is identical to FP for PCE without equality, we use FP in this section.

Our approach can be illustrated by the network given in Figure 5.2, where the constraints are specified with each node and edge. For example, the value of variable A could be either 1 or 3, and variables A and B can together take values 1 and 4, or 3 and 6, respectively.

Consider the following line of reasoning: suppose variable A is assigned the value 3. It follows from the constraint on the edge AB that B's value should be 6, and from the constraint on the edge AC that C's value should be 9. Since these values of B and C are inconsistent with the constraint on the edge BC, the value of A cannot be 3; perhaps it could be 1. By propagating A's value of 1 through the constraint on AB, we obtain the value 3 for B, and by propagating A's value through the constraint on AC, we obtain that C's value could be either 7 or 8. The results of these two cases of hypothetical reasoning can be recorded in a "simpler" network, Figure 5.3, which has the same solutions as the original one. An important point is that in obtaining the new network we never had to make more than one assumption.

This kind of reasoning can be captured using the fixpoint construction based on fact propagation (Section 4.4) applied to a theory obtained from the above network. This theory contains formulas which are built from atoms like $A{:}3$, denoting the assignment of value 3 to node A — formulas that encode the various constraints of the network. For example, the constraint on edge AB is encoded by the formula

$$((A{:}1 \wedge B{:}4) \vee (A{:}3 \wedge B{:}6))$$

while the restriction on node A is encode by $(A{:}1) \vee (A{:}3)$. There is also a formula expressing the fact that A can only be assigned one value: $\neg(A{:}1) \vee \neg(A{:}3)$.

Assigning value 3 to variable A in the network corresponds to adding the formula $A{:}3$ to the translated theory; the resulting inconsistency in the network corresponds to obtaining $\{\mathbf{f}\}$ from the augmented theory using fact propagation. Since the fixpoint construction works on the basis of refutation, it then follows that the clause $(\neg A{:}3)$ is in the fixpoint. Fact propagation then "pushes" this through the constraint $(A{:}1) \vee (A{:}3)$, yielding that $A{:}1$ is in the fixpoint. Similarly, one argues that the clauses $(B{:}4)$, and $(\neg C{:}9)$ are also in the fixpoint. It turns out that the translated theory augmented by the fixpoint corresponds exactly to the network of Figure 5.3.

Note that we have considered only clauses of size 1 in constructing this fixpoint from the translated theory; this corresponds to making only a single level of assumptions (i.e., no nested assumptions) in reasoning with the original network. The significance of this is that if a network is inconsistent, and we can discover the inconsistency by making assumptions of size at most 1, then its theory has intricacy 1.

5.4.2 Tractable Constraint Networks

This section provides formal definitions concerning constraint networks and reviews some previously-proposed classes of tractable constraint networks. For a more detailed description, the interested reader is referred to [Mac87, Dec91, DH91].

Since we will be interested in families of constraint networks, we start with a denumerable set $\mathcal{V}$ of *values*, and a denumerable set $\mathcal{X}$ of *variables*, sometimes called *nodes*. (Despite their name, these are not to be confused with the variables of predicate calculus.)

Definition 5.4 A *constraint network*, $C = (X, V, E, c)$, consists of a finite set X of variables from $\mathcal{X}$, a finite subset V of values from $\mathcal{V}$, a set $E \subseteq X \times X$ of edges, and a partial function c from the set of k-tuples of variables in X to the powerset of k-tuples of values in V, for $1 \leq k \leq |X|$. There are several restrictions on the partial function c of constraints:[2]

1. $c(x)$ is defined and is non-empty for each variable x in X; $c(x)$ is called the *domain* of x.
2. If $(v_1, \ldots, v_r) \in c(x_1, \ldots, x_r)$ for variables $x_1, \ldots, x_r$ and values $v_1, \ldots, v_r$, then $v_i \in c(x_i)$ and all variables in the set $\{x_1, \ldots, x_r\}$ are distinct.
3. If $c(x_1, \ldots, x_r)$ is defined then the graph (X, E) is complete on the nodes $\{x_1, \ldots, x_r\}$ (r is called the *arity* of this constraint).

A *valuation* is a *partial* function θ from the variables in X to values in V such that $\theta(x) \in c(x)$ (i.e., domain constraints are already satisfied) for those x for which θ is defined. A valuation θ over the entire set of variables X is a *complete valuation*. A *solution* is a complete valuation θ such that all the constraints in c are satisfied: $(\theta(x_1), \ldots, \theta(x_r)) \in c(x_1, \ldots, x_r)$ for each tuple $(x_1, \ldots, x_r)$ of variables for which c is defined.

For any set Y of variables, the *restriction* C/Y is the network $(X \cap Y, V, E \cap (Y \times Y), c')$ where c' is c restricted to the domain of tuples over $X \cap Y$. Any solution of a restricted network is called a *partial solution* of the network. ■

[2] Since c is a unary function, $c((x_1, \ldots, x_n))$ is abbreviated as $c(x_1, \ldots, x_n)$, as in some functional programming languages.

For example, consider a CSP network $C1 = (X, V, E, c)$ defined as follows:

$$X = \{a, b, d\}$$
$$E = \{(a, d), (b, d)\}$$
$$V = \{4, 5, 6, 7, 8, 9\}$$
$$c(a) = \{4, 5\}; c(b) = \{6, 7\}; c(d) = \{8, 9\}$$
$$c(a, d) = \{(4, 9), (5, 8), (5, 9)\}$$
$$c(b, d) = \{(6, 8), (7, 8), (7, 9)\}$$

A valuation that assigns values 4, 7, and 9 to nodes a, b, and d respectively, is a solution of this network. There are many other solutions, as well. However, a valuation that assigns values 4, 6, and 9 to nodes a, b, and d respectively, is not a solution, since it violates the constraint $c(b, d)$.

The following three general observations may be useful to keep in mind:

1. In general, the topology of the graph (X, E) indicates the kinds of constraints allowed in the network, but it does not completely capture this information; for example, the edges $(d, e), (e, g), (g, h)$ together could denote either a ternary constraint among the variables d, e, h, or three binary constraints corresponding to the three edges, or both.

2. Each constraint is a set of values allowed for a particular tuple of variables, so the constraint $c(x_1, \ldots, x_r) = \{(v_1^i, \ldots, v_r^i) \mid 1 \leq i \leq k\}$ for some k corresponds to a collection of valuations $\{s_i \mid 1 \leq i \leq k,\ s_i(x_1) = v_1^i, \ldots, s_i(x_r) = v_r^i\}$. Note that each constraint could also have been specified by listing its complement – the valuations over the variables $x_1, \ldots, x_r$ that are not allowed / "ruled out".)

3. Traditionally, in CSP research, one starts with *binary constraint networks* — networks in which all constraints are unary or binary. Unfortunately, as we shall see, the processing of such networks may introduce constraints of higher arities.

Backtrack-free Networks

Algorithms for finding solutions of constraint networks typically search through the space of all valuations generated by the cross-product of the variable domains. The solutions are built incrementally by assigning values to variables in some fixed order and verifying that all the constraints among the variables assigned values so far are satisfied. If some constraint is violated, then a new value is assigned to the current variable; if all values allowed by the unary constraint of the current variable violate the constraints then the value assigned to the previous variable is changed, causing *backtracking.*

For example, suppose the variables of the network $C1$ are ordered a, b, d. Suppose a is first assigned the value 4, and b is then assigned the value 6. Since either value of d, together with the assigned values of a and b, violates some constraint, the search backtracks to b, which is now assigned the value 7. Now d can be assigned the value 9 without violating any constraint, leading to the solution mentioned above.

Now consider a new binary network $C1'$ which is obtained from $C1$ by adding the following constraint:

$$c(a, b) = \{(4, 7), (5, 6), (5, 7)\},$$

and correspondingly changing the set of edges to:

$$E' = \{(a, b), (a, d), (b, d)\}.$$

Adding this constraint prevents the assignment of value 6 to variable b, after 4 has been assigned to a. Thus, the above solution of $C1$ is also obtained as a solution of $C1'$, but *without* any backtracking. It can be verified that $C1$ and $C1'$ have identical solutions, which can be obtained in $C1'$ without any backtracking, when using the ordering a, b, d. Thus, $C1'$ is said to be a *backtrack-free* network with respect to this ordering.

All backtrack-free networks are consistent by definition, and it is known that any consistent binary network can be transformed into an equivalent (i.e., having the same solutions) backtrack-free network with respect to any given ordering by adding new constraints [Dec91]. (The transformation from Figure 5.2 to Figure 5.3 is an example of such a transformation.) It is important to remember that, as illustrated later, these new constraints may be of arbitrary arity even if we start with a binary constraint network.

The Consistency Problem

Instead of finding the solutions of a constraint network, we are interested in a restricted problem: determining whether the given network is consistent.

Definition 5.5 The *consistency problem* for a class $\mathcal{C}$ of binary constraint networks is given by:

Input:	a network $C \in \mathcal{C}$;
output:	"yes" iff there is a solution of C.

A class $\mathcal{C}$ of binary constraint networks is *tractable* iff the consistency problem for $\mathcal{C}$ is in PTIME. ∎

Since the consistency problem for the class of all binary constraint networks is CoNP-Complete [Mac87], identifying classes of constraint networks for which consistency is provably tractable has generated considerable interest (c.f. [Fre82, DP87, DH91]). Most techniques to identify tractable families of CSP rely on the topology of the underlying constraint network. For example, Freuder [Fre82, Fre85] observed that CSPs whose networks are trees can be solved in linear time.

Dechter and Pearl [DP87] defined a topological property of networks called *induced width*, and showed that for any network for which there is a number k that bounds the induced width, it is possible to determine the consistency of the network in time polynomial in its size but exponential in k. The family of "bounded width" networks — ones whose induced width is bounded by some fixed constant k — is, so far, the largest family of tractable constraint networks identified by "topological criteria".

As part of the above, Dechter and Pearl proposed an algorithm called *Adaptive-Consistency* (described below) which, among others, determines the inconsistency of binary constraint network.

Adaptive Consistency Algorithm

Given two arguments, a binary constraint network **and** an ordering of its variables, the *Adaptive-Consistency* algorithm [DP88][Page 18] reports failure iff the network is inconsistent. (Otherwise, it returns an equivalent backtrack-free network, though this will not be of concern to us here; nor will the efficiency of this algorithm.)

As an illustration, given the network $C1$ and the ordering a, b, d, *Adaptive-Consistency* first processes the variable d and adds the constraint $c(a, b)$ mentioned above. The algorithm generates this constraint by considering all valuations over the nodes that precede d (i.e., a and b), and *ruling out* those which are not consistent with any value of d; so in this example, the valuation assigning $(4, 6)$ to (a, b) is ruled out.[3] The algorithm then processes b, and since this causes no further changes, the (backtrack-free) network $C1'$ is returned as the output.

Essentially, *Adaptive-Consistency* processes the nodes in reverse order and rules out valuations (among nodes ordered earlier than the current node) that cause backtracking. The time complexity of the algorithm is exponential in the maximum arity, say, r, of the constraints that are found or generated, because $|V|^r$ possible valuations over those r variables may be considered.

A key idea in the algorithm is to minimize this number r by using the observation that only a restricted set of nodes needs to be considered in generating the new constraints. In particular, while processing any node x only those nodes are considered that are ordered earlier than x *and* share an edge with x in the *current state* of the network; therefore, any new constraint added while processing x involves no more nodes than these.

[3] Note that adding such a constraint results in the graph of the new network having an additional edge (a, b).

(a) Network (b) Induced Network

Figure 5.4: Induced Network

Interestingly, a superset of the edges of the output network can be easily determined without even executing the algorithm, by executing the following step for every node z in the reverse of the given ordering: if there are edges (x, z) and (y, z) such that both x and y are ordered before z then add the edge (x, y). The rationale is that only constraints involving x and y might be generated by the algorithm while processing the node z. The network obtained by adding edges in this way is called the induced network with respect to the given ordering. For example, consider the network (X, V, E, c) whose graph is shown in Figure 5.4 (a), where $X = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (a, c), (a, d), (b, e), (b, c), (d, e), (e, f)\}$. The graph of the induced network for the ordering $a < b < c < d < e < f$ is given in Figure 5.4 (b).

Formally, given any binary constraint network $C = (X, V, E, c)$ and any total order $<$ on the set X, the ***induced network*** $C^\star$ with respect to $<$ is the network $(X, V, E^\star, c)$, where $E^\star$ is the least superset of E such that if $(x, z), (y, z) \in E^\star$ and $x < z$ and $y < z$ then $(x, y) \in E^\star$. For any variable x in X, parents(x) in C is defined to be the set $\{y \in X \mid y < x, (x, y) \in E\}$. *Adaptive-Consistency* processes the nodes in decreasing order of $<$. For node x, the algorithm calls the procedure *Consistency*(x, parents(x)), which rules out all valuations over all variables of parents(x) in $C^\star$ that are not consistent with some value of x, and it considers only those constraints all of whose variables are from the set parents(x)$\cup\{x\}$. (Note that since only constraints involving *parents*(x) are added to the new network, the sets *parents*(y) do not change for $y = x$ or nodes y examined before x.) [DP88] proves that the network C is unsatisfiable iff there is a node x such that all valuations over parents(x) are ruled out during the call to *Consistency*(x, parents(x)). Otherwise, the output network is backtrack-free.

To provide insight into the *Adaptive-Consistency* algorithm and to illustrate the generation of non-unary and non-binary constraints, we give two examples: first a consistent network and then an inconsistent one. Consider a network $C2 = (X, V, E, c)$ defined as follows:

$$
\begin{gathered}
X = \{p, q, r, s\} \\
V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \eta\} \\
E = X \times X \\
c(p) = dom(p) = \{1, 2\} \\
c(q) = dom(q) = \{3, 4, 5\} \\
c(r) = dom(r) = \{6, 7, 8\} \\
c(s) = dom(s) = \{9, 0, \eta\} \\
c(p, q) = \{(1, 3), (1, 4)\} \\
c(p, r) = c(p) \times c(r) - \{(1, 7)\} \\
c(p, s) = c(p) \times c(s) - \{(1, 9)\} \\
c(q, r) = c(q) \times c(r) - \{(3, 8), (4, 6)\} \\
c(q, s) = c(q) \times c(s) - \{(3, 0)\} \\
c(r, s) = c(r) \times c(s) - \{(6, \eta)\}
\end{gathered}
$$

Suppose *Adaptive-Consistency* processes nodes in the order s, r, q, p (i.e., the order $<$ is the usual lexicographic ordering). Note that the induced network $C2^\star$ is identical to $C2$. While processing s, *Consistency*($s, \{p, q, r\}$) rules out the valuation that assigns 1, 3 and 6 to p, q, and r, respectively, because each extension of this valuation to node s is disallowed by a constraint among some parents of s ($c(p, s)$ disallows 9, $c(q, s)$ disallows 0, and $c(r, s)$ disallows η). It also rules out many other valuations (such as w where $w(p) = 2$, $w(q) = 4$, $w(r) = 6$) that violate the original constraints ($c(p, q)$ in this case). As a result of these "rulings out", the new constraint generated is $c(p, q, r) = \{(1, 4, 7), (1, 4, 8)\}$. While processing r, *Consistency*($r, \{q, p\}$) rules out the valuation that assigns 1 and 3 to p and q, respectively. (Note that while this valuation is also implicitly ruled out by the constraint $c(p, q, r)$, the constraint cannot be used when only the variables p and q are being considered in the backtrack-free search.) It also rules out assignments like 1 and 5 to p and q, respectively, which are already ruled out by some original constraint. While processing q, *Consistency*($q, \{p\}$) rules out the valuation that assigns 2 to p. The resulting network, which is returned by *Adaptive-Consistency* is backtrack-free with two solutions: $1, 4, 8, 0$ and $1, 4, 8, \eta$ assigned to p, q, r, s, respectively.

Consider another network $C2' = (X, V, E, c')$, which is identical to $C2$ except that $c'(q, r) = c(q, r) - \{(4, 8)\}$. Suppose *Adaptive-Consistency* processes the nodes again in the order s, r, q, p. *Consistency*(s,{p,q,r}) rules out the same valuations as before. However, *Consistency*($r, \{q, p\}$) now rules out all valuations over $\{p, q\}$. Thus, $C2'$ is declared inconsistent by *Adaptive-Consistency*.

The following is a summary of observations about *Adaptive-Consistency* that may be useful to keep in mind:

1. The input consists of a *binary* network and a total ordering of its nodes.
2. In dealing with a constraint network, one can just as well think of a constraint in terms of the valuations ruled out by it, as the valuations permitted by it.
3. However constraints are represented, the algorithm works by ruling out valuations: for a node x, valuations involving parents(x) $\cup \{x\}$ are considered, and valuations involving only variables in the set parents(x) are ruled out.
4. Although the input network is binary, the valuations ruled out may contain more than two variables.
5. A network is inconsistent if and only if there is a node x for which the algorithm rules out all valuations over the variables in parents(x).

Induced Width

Recall that while processing a node x in a binary network $C = (X, V, E, c)$ with some total order $<$ on the set X, the *Adaptive-Consistency* algorithm calls *Consistency*(x, parents(x)), which considers at most those valuations all of whose variables are among x and parents(x) in $C^\star$. Since the time complexity of the algorithm is exponential in the maximum number of variables in the valuations that are considered by the various calls to *Consistency*, we desire an ordering which minimizes the size k of the set parents(x) for any x in $C^\star$. This number k is the "induced width" of the network, and is defined formally as follows:

Definition 5.6 For any network $C = (X, V, E, c)$ and any total order $<$ on X, the *width of a variable* $x \in X$ with respect to $<$ is the cardinality of the set parents(x). The *width of* C *with respect to* $<$ is the maximum of the widths of the variables in X with respect to $<$ in C. The *width* of the network C is the minimum width of C over all total orders on X. The *induced width of* C *with respect to* $<$ is the width of $C^\star$ with respect to $<$. The *induced width* of C is the minimum of induced widths of C over all total orders on X. ■

For example, consider the network shown in Figure 5.4(a) and the ordering $a < b < c < d < e < f$. Note that parents(e) $= \{b, d\}$, since $f > e$. It can be verified that the width with respect to $<$ of a is 0, b is 1, c is 2, d is 1, e is 2, and f is 1. Thus, the width of the network in (a) with respect to $<$ is 2. Recall that the induced graph with respect to this ordering is given in Figure 5.4(b). Except for d, whose width is 2, the widths of various nodes in (b) is the same as in (a). Thus, the induced width of the network in (a) with respect to $<$ is also 2.

Our eventual goal is simply to prove that if the induced width of an inconsistent network C is k, then the corresponding theory has intricacy bounded by $k+1$. For this reason, we will not be concerned with finding the ordering that minimizes the induced width, and will assume that it is given by hypothesis; for the same reason, we will not be concerned with the time complexity of algorithms *Consistency* or *Adaptive-Consistency*.

Although not relevant for the remainder of this section, we remark that the class of constraint networks with induced width bounded by k was, of course, known to be tractable: Given some ordering of the nodes, the total time taken by the calls to *Consistency* determines the worst-case time complexity of *Adaptive-Consistency*, which then is polynomial in the size of the network, but exponential in its induced width with respect to the given ordering. And, given a binary constraint network that is known to have induced width bounded by k, [Arn85, Dec91] provide tractable algorithms for finding an ordering that achieves this bound, and hence can be passed to *Adaptive-Consistency*. Also, we observe that Arnborg [Arn85] showed that determining the induced width of any specific graph is an NP-complete problem, so it is not practical to determine it "at run time". However, it has been argued in [Dec91] that the bounded induced width criterion can be used as a theoretical tool for identifying tractable families of CSPs.

Functional Constraints

There have been other approaches for identifying tractable instances of CSP. Deville and Van Hentenryck [DH91] use the "semantics" of the constraints to identify a new tractable class of CSPs. A constraint $c(x, y)$ is said to be *functional* iff for each value in $c(x)$, there is at most one value in $c(y)$ (and vice versa) that satisfies the constraint. [DH91] presents a quadratic-time algorithm for determining consistency of binary networks where *all* the constraints are functional. (The algorithm also performs "arc-consistency" [Fre85] for consistent networks in this class.) An example of a network in this class can be obtained from the network of Figure 5.2 by removing the tuple $(1, 8)$ from the constraint $c(A, C)$, and the tuples $(4, 7)$ and $(4, 8)$ from the constraint $c(B, C)$.

5.4.3 Constraint Theories

As discussed in Section 5.4.1, we wish to prove results about the intricacy of certain logical theories that "represent" constraint networks. In this section we will formally relate constraint networks to specific kinds of propositional theories, and show how valuations of the constraint networks relate to interpretations of the corresponding constraint theories. This prepares the way for the next section where we use these to show the tight connection between the induced width of an inconsistent constraint network and the intricacy of its corresponding propositional theory.

Recall that we have a denumerable set $\mathcal{V}$ of values, and a denumerable set $\mathcal{X}$ of variables. The only *atoms* in our language are from the set $\{x{:}v \mid x \in \mathcal{X}, v \in \mathcal{V}\}$. For any formula ψ in the PC language generated using these atoms, the set of variables in ψ is given by:

$$vars(\psi) = \{x \in \mathcal{X} \mid \exists v \in \mathcal{V} \text{ s.t. } x{:}v \in \text{atoms}(\psi)\}$$

For any theory Γ, the set of variables in Γ is given by:

$$vars(\Gamma) = \cup\{vars(\psi) \mid \psi \in \Gamma\}$$

Recall from Section 2.3.2 that an interpretation of a theory is a mapping from its atoms to the set $\{true, false\}$, and that a model is an interpretation which maps the theory to $true$. Since our aim is to relate constraint networks to theories, we wish to associate *valuations that are solutions* of constraint networks with *models* of the corresponding theories. Intuitively, a complete valuation θ over X can be associated with an interpretation that maps $x{:}\theta(x)$ to $true$ for each x in X, and maps all other atoms to $false$. Thus, we need to a priori rule out any interpretation that, for some variable x, either maps each $x{:}v$ to $false$ or maps some $x{:}v_1$ and $x{:}v_2$ to *true* where $v_1 \neq v_2$. This is done by adding some special formulas to our theories. For this, we associate from the beginning with each variable a finite non-empty set of values called its *domain*; formally, a domain function $Dom : X \rightarrow 2^V$ has the property that for any variable $x \in X$, $Dom(x)$ is finite and non-empty.

We now proceed to overload the term "solution" (and very slightly "valuation") so it also applies to certain logical theories.

Definition 5.7 Given sets X and V and domain function Dom, the *domain constraints* for any variable $x \in X$ are the clauses $(x{:}v_1 \vee \ldots \vee x{:}v_n)$ and $(\neg x{:}v_i \vee \neg x{:}v_j)$, where $i, j \in 1 \ldots n$, $i \neq j$, and $Dom(x) = \{v_1, \ldots, v_n\}$. A *constraint theory* Γ over X, V and Dom is a theory whose formulas involve only literals of the form $x : v$, for $x \in X$, $v \in Dom(x)$, and include all the domain constraints for all variables in the theory.

For any set $Y \subseteq X$, a *valuation* over Y is a function θ from Y to V such that $\theta(y) \in Dom(y)$ for each $y \in Y$. Any valuation over $vars(\Gamma)$ is called a *complete valuation* of Γ. Any valuation whose domain is $vars(\Gamma)$ is called a *complete valuation* of Γ. For any complete valuation θ of Γ, the interpretation I_θ of Γ is defined by:

$$I_\theta(x{:}v) = \begin{cases} true & \text{if } v = \theta(x) \\ false & \text{if } v \in Dom(x) - \{\theta(x)\} \end{cases}$$

for each $x \in X$ and each $v \in Dom(x)$. A *solution* of Γ is a complete valuation θ such that I_θ is a model of Γ (we also say that θ *solves* Γ). ∎

Note that, because of the presence of domain constraints, for any model M of a constraint theory Γ there is exactly one valuation θ over $vars(\Gamma)$ such that M maps an atom $x{:}v$ to *true* iff $\theta(x) = v$. The atoms mapped to *false* in M are $x{:}w$ where $x \in X$ and $w \in Dom(x) - \{\theta(x)\}$.

These valuations, which correspond to models of a constraint theory, are the solutions of the constraint theory.

We can also associate with every valuation θ over variables Y a conjunctive formula $\widehat{\theta} = (x_1{:}v_1 \wedge \ldots \wedge x_n{:}v_n)$, where $Y = \{x_1, \ldots, x_n\}$ and $\theta(x_i) = v_i$ for $i = 1 \ldots n$; thus, $\sim\widehat{\theta} = (\neg x_1{:}v_1 \vee \ldots \vee \neg x_n{:}v_n)$ is a basic clause, called the *constraint clause* corresponding to valuation θ. (If Y is empty then $\widehat{\theta} = \mathbf{t}$ and $\sim\widehat{\theta} = \mathbf{f}$.) The intuition, which we shall use later, is that such a constraint clause $\sim\widehat{\theta}$ rules out valuation θ as a partial solution, when added to a constraint theory.

It follows from the above definitions and observations that a complete valuation θ is a solution of a constraint theory Γ iff $\Gamma \cup \{\widehat{\theta}\}$ is satisfiable iff $\Gamma \not\models \sim\widehat{\theta}$. We shall make repeated use of these equivalences later.

Translation of Binary Networks to Theories

Note that valuations for a constraint theory are identical to valuations for a constraint network if the theory and the network have identical sets of variables X and values V, and Dom is the unary constraint of the network.

We now show that each binary constraint network can be translated into a constraint theory such that the two have identical solutions.

Definition 5.8 For any binary network $C = (X, V, E, c)$, its *translation* $\mathrm{Tr}(C)$ is the theory over X, V and Dom (where $Dom(x) = c(x)$ for every $x \in X$) containing exactly the following formulas:

(Type A): $(x{:}v_1 \vee \ldots \vee x{:}v_n)$ for each $x \in X$ such that $c(x) = \{v_1 \ldots v_n\}$;

(Type B): $(\neg x{:}v_i \vee \neg x{:}v_j)$ for each $x \in X$ such that $c(x) = \{v_1 \ldots v_n\}$, $i, j \in 1 \ldots n$, and $i \neq j$;

(Type D): $((x{:}v_1 \wedge y{:}w_1) \vee \ldots \vee (x{:}v_m \wedge y{:}w_m))$ for each $(x, y) \in E$ such that $\{(v_1, w_1), \ldots, (v_m, w_m)\} = c(x, y)$.[4]

[4] There are no "Type C" formulas, so that there is no confusion with "C" used for a constraint network.

■

It follows that for any binary network $C = (X, V, E, c)$, $\mathrm{Tr}(C)$ is a constraint theory (over X, Y and Dom, which we shall omit in such cases, since they are implicit in C.) Note that since $Dom(x) = c(x)$, valuations of a constraint network are entirely identical to valuations of the corresponding constraint theory $\mathrm{Tr}(C)$.

For example, the translation $\mathrm{Tr}(C1)$ of the network given above contains the following formulas:

$$
\begin{array}{cc}
\multicolumn{2}{c}{((a{:}4 \wedge d{:}9) \vee (a{:}5 \wedge d{:}8) \vee (a{:}5 \wedge d{:}9))} \\
\multicolumn{2}{c}{((b{:}6 \wedge d{:}8) \vee (b{:}7 \wedge d{:}8) \vee (b{:}7 \wedge d{:}9))} \\
(a{:}4 \vee a{:}5) & (\neg a{:}4 \vee \neg a{:}5) \\
(b{:}6 \vee b{:}7) & (\neg b{:}6 \vee \neg b{:}7) \\
(d{:}8 \vee d{:}9) & (\neg d{:}8 \vee \neg d{:}9)
\end{array}
$$

It can be verified that the interpretation that assigns *true* to only $a{:}4$, $b{:}7$, and $d{:}9$ is a model of $\mathrm{Tr}(C1)$, and that the interpretation that assigns *true* to only $a{:}4$, $b{:}6$, and $d{:}9$ is not.

Lemma 5.7 shows that, not surprisingly, the solutions of any binary constraint network are exactly the solutions of the corresponding constraint theory.

Lemma 5.7 *Any valuation θ solves a binary constraint network C iff θ solves* $\mathrm{Tr}(C)$.

Proof:

Consider any binary network $C = (X, V, E, c)$ and any valuation θ over X. Recall that $Dom(x) = c(x)$ for any $x \in X$.

θ solves C	iff	θ satisfies each constraint in C
	iff	(1) $\forall x \in X, \theta(x) \in c(x)$, and
		(2) $\forall (x, y) \in E, (\theta(x), \theta(y)) \in c(x, y)$
	iff(*)	I_θ is a model of $\mathrm{Tr}(C)$
	iff	θ solves $\mathrm{Tr}(C)$.

We now argue for the only non-trivial step above, which is marked by (*). For the "only if" direction, it can be verified that I_θ is a model of each formula in $\mathrm{Tr}(C)$: for formulas of types (A) and (B) because of (1), and of type (D) because of (2). For the "if" direction, it can be verified that θ satisfies (1) because of type A formulas and satisfies (2) because of type D formulas. ■

We now define restrictions of a constraint theory, analogous to restrictions for constraint networks.

Definition 5.9 The ***restriction*** Γ/Y of a constraint theory Γ to a set Y of variables is the theory $\{\psi \in \Gamma \mid vars(\psi) \subseteq Y\}$. A valuation θ is a ***partial solution*** of a theory Γ iff θ solves $\Gamma/vars(\theta)$. Valuation θ' is an ***extension*** of valuation θ iff $\sim\widehat{\theta}$ is a subclause of $\sim\widehat{\theta'}$. ■

It follows that a complete valuation is a solution iff it is a partial solution. Lemma 5.8 shows that restrictions on constraint networks also correspond exactly to the restrictions on the constraint theories.

Lemma 5.8 *For any binary constraint network C and any set Y of variables:* $\mathrm{Tr}(C/Y) = \mathrm{Tr}(C)/Y$.

Proof: Consider any binary network $C = (X, V, E, c)$ and any set Y of variables. Recall that $Dom(x) = c(x)$ for any $x \in X$, and that $C/Y = (X \cap Y, V, E \cap (Y \times Y), c')$, where c' is c restricted to the domain of tuples over $X \cap Y$. Any formula ψ in $\mathrm{Tr}(C/Y)$ has one of the following types:

Type A: $\psi = (x{:}v_1 \vee \ldots \vee x{:}v_n)$ such that $x \in X \cap Y$ and $c(x) = \{v_1 \ldots v_n\}$;

Type B: $\psi = (\neg x{:}v \vee \neg x{:}w)$ such that $x \in X \cap Y$, $v, w \in c(x)$, and $v \neq w$;

Type D: $\psi = ((x{:}v_1 \wedge y{:}w_1) \vee \ldots \vee (x{:}v_m \wedge y{:}w_m))$ such that $(x, y) \in E \cap (Y \times Y)$ and $\{(v_1, w_1), \ldots, (v_m, w_m)\} = c(x, y)$.

In each of the above cases, $\psi \in \mathrm{Tr}(C/Y)$ iff ($\psi \in \mathrm{Tr}(C)$ and $vars(\psi) \subseteq Y$) iff $\psi \in \mathrm{Tr}(C)/Y$. Thus, $\mathrm{Tr}(C/Y) = \mathrm{Tr}(C)/Y$. ∎

It follows directly from the above two lemmas that partial solutions for networks and theories are also identical: for any binary constraint network C, any valuation θ over any set Y of variables solves the network C/Y iff θ is a partial solution of the theory $\mathrm{Tr}(C)$.

5.4.4 Induced Width and Intricacy

We now show that the induced width of any inconsistent binary network is always greater than $m-1$, where m is the intricacy of its translated theory.

To prove this, we will make use of the correctness of the *Adaptive-Consistency* algorithm. Recall that if *Adaptive-Consistency* uses an ordering that realizes the induced width of a binary network, then the number of variables in the largest valuation that is ever ruled out is no more than the induced width of the network. An important result in this section will be Lemma 5.10, showing that if *Adaptive-Consistency* rules out a valuation θ over k variables then the constraint clause $\sim\widehat{\theta}$, which is of size k, is in the $(k+1)$-th least fixpoint for the translated theory. The main result will be Theorem 5.11, arguing that since *Adaptive-Consistency* rules out all valuations over some set of variables for an inconsistent network, it follows that $\mathbf{f}$ is in the fixpoint; thus, the intricacy is at most $(k+1)$. For this, we need the following lemma (5.9), which shows that $\vdash_{\mathrm{FP}}$ can be used to verify solutions of a constraint theory:

Lemma 5.9 *For any constraint theory* Γ *and any complete valuation* θ *for* Γ*,* θ *solves* Γ *iff* $\Gamma \not\vdash_{\mathrm{FP}} \sim\widehat{\theta}$.

Proof: Consider a constraint theory Γ over X, V, Dom, and a complete valuation θ over Γ. Suppose S_θ is the set of all literals assigned *true* in the interpretation I_θ, i.e., the set $\{x{:}\theta(x) \mid x \in X\} \cup \{\neg x{:}v \mid x \in X, v \in Dom(x), v \neq \theta(x)\}$, and formula ψ_θ is obtained by taking the conjunction of all literals in S_θ.

Consider the theory $\Gamma \cup \{\widehat{\theta}\}$. It follows from modularity and Property C† that $\Gamma \cup \{\widehat{\theta}\} \Leftrightarrow^*_{\mathrm{FP}} \Gamma \cup \{x{:}\theta(x) \mid x \in X\}$, and then from properties D† and E† that $\Gamma \cup \{\widehat{\theta}\} \Leftrightarrow^*_{\mathrm{FP}} \Gamma \cup S_\theta$ (because Γ contains type B formulas $\neg x{:}v \vee \neg x{:}\theta(x)$). Thus, $\Gamma \vdash_{\mathrm{FP}} \sim\psi_\theta$ iff $\Gamma \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Since $\mathrm{atoms}(\Gamma) \subseteq \mathrm{atoms}(\sim\psi_\theta)$, it then follows from Proposition 4.6 and soundness of $\vdash_{\mathrm{FP}}$ that $\Gamma \models \sim\psi_\theta$ iff $\Gamma \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Therefore,

$$
\begin{array}{lll}
\Gamma \not\vdash_{\mathrm{FP}} \sim\widehat{\theta} & \text{iff} & \Gamma \not\models \sim\psi_\theta \\
 & \text{iff} & \Gamma \cup \{\psi_\theta\} \text{ is satisfiable} \\
 & \text{iff} & I_\theta \text{ is a model of } \Gamma \\
 & \text{iff} & \theta \text{ solves } \Gamma
\end{array}
$$

∎

Before presenting Lemma 5.10, we will show an example of the kind of argument that will be used in its proof. Consider the network $C2$ given above, whose induced width is 3 with respect to order $p < q < r$, and the valuation θ assigning $1, 3, 6$ to p, q, r, which is ruled out by *Adaptive-Consistency* while processing node s. We will now argue that the constraint clause $\sim\widehat{\theta} = \sim(p{:}1 \wedge q{:}3 \wedge r{:}6)$ is in the fixpoint $\mathrm{lfp}(T_{\mathrm{Tr}(C2),4})$. Since no extension θ' of θ to node s is a partial solution, it follows from Lemma 5.9 that $\mathrm{Tr}(C2) \vdash_{\mathrm{FP}} \sim\widehat{\theta'}$ for each such θ'. Since each $\sim\widehat{\theta'}$ is also a basic clause of size s, it is in the fixpoint. We then obtain from Proposition 4.8 that $\mathrm{Tr}(C2) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Thus, the constraint $\sim\widehat{\theta}$ is also in the fixpoint.

Since the induced width of the network C2 with respect to the ordering $<$ is 3, *Adaptive-Consistency* only needs to rule out valuations of size at most 3. Thus, our fixpoint construction needs to consider only valuations of size one larger, namely 4.

Lemma 5.10 *For any binary constraint network* C *of induced width* k*, if* Adaptive-Consistency *rules out a valuation* θ *then* $\sim\widehat{\theta} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$.

Proof: Let $C = (X, V, E, c)$ be any binary constraint network of induced width k, and suppose $<$ is the ordering on X that achieves it.

Let Γ be the theory $\mathrm{Tr}(C)$ and Δ be the theory $\mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$. We will prove the claim of the lemma by contradiction.

Suppose the claim is false; let θ be the first valuation ruled out by *Adaptive-Consistency* such that $\sim\widehat{\theta} \notin \Delta$. Let $x \in X$ be the node being processed by *Adaptive-Consistency* while ruling out θ, let ψ be the constraint clause $\sim\widehat{\theta}$, and let Γ' be the set of all constraint clauses $\sim\widehat{\theta'}$ such that the valuation θ' is ruled out by *Adaptive-Consistency* before processing x. It follows that $\Gamma' \subseteq \Delta$.

Since the ordering used by *Adaptive-Consistency* realizes the induced width k of C, valuation θ is over k nodes. Thus, the size of the constriant clause ψ is also k.

Since *Adaptive-Consistency* rules out the valuation θ while processing node x, for no $v \in Dom(x)$ is the valuation θ'' which extends θ by assigning v to x a partial solution of the modified network, i.e., when considering only the constraints on $\mathrm{parents}(x) \cup \{x\}$. Hence $\Gamma \cup \Gamma'$ must entail $(\psi \mathbin{\mathring{\vee}} \neg x{:}v)$, or more precisely $(\Gamma \cup \Gamma')/(\mathrm{parents}(x) \cup \{x\}) \models (\psi \mathbin{\mathring{\vee}} \neg x{:}v)$. Using Lemma 5.9, we obtain that $(\Gamma \cup \Gamma')/(\mathrm{parents}(x) \cup \{x\}) \vdash_{\mathrm{FP}} (\psi \mathbin{\mathring{\vee}} \neg x{:}v)$, and then from Proposition 4.3 that $\Gamma \cup \Delta \vdash_{\mathrm{FP}} (\psi \mathbin{\mathring{\vee}} \neg x{:}v)$. That is, $(\psi \mathbin{\mathring{\vee}} \neg x{:}v) \in T_{\Gamma,k+1}(\Delta)$. Since Δ is the least fixpoint of $T_{\Gamma,k+1}$, we obtain that $(\psi \mathbin{\mathring{\vee}} \neg x{:}v) \in \Delta$.

As the above holds for each $v \in Dom(x)$, it follows from Proposition 4.8 that $\Delta \vdash_{\mathrm{FP}} \psi$. (Intuitively, $\sim\psi$ "cancels out" ψ in $(\psi \mathbin{\mathring{\vee}} \neg x{:}v)$, leaving $\neg x{:}v$ for every $v \in Dom(x)$, which in turn contradicts the domain assertion

$$(x{:}v_1 \vee \ldots \vee x{:}v_n) \text{ where } c(x) = \{v_1 \ldots v_n\}$$

for variable x in the theory $\mathrm{Tr}(C)$. As a result, FP reduces $\Delta \cup \{\sim\psi\}$ to $\{\mathbf{f}\}$.)

Since Δ is a fixpoint, $\Delta \vdash_{\mathrm{FP}} \psi$ entails $\psi \in \Delta$, which is a contradiction.

■

Theorem 5.11 *For any inconsistent binary network C of induced width k, the intricacy of* $\mathrm{Tr}(C)$ *is at most $k+1$.*

Proof: Suppose $C = (X, V, E, c)$ is an inconsistent binary constraint network of induced width k, realized with ordering $<$. All we need to show is that $\mathbf{f} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$, i.e., $\mathrm{Tr}(C)$ is not $(k+1)$-consistent. Since C is inconsistent, there is a variable $x \in X$ such that each valuation θ over $\mathrm{parents}(x)$ is ruled out by *Adaptive-Consistency*. Since each such valuation is defined over at most $k+1$ variables, $\sim\widehat{\theta}$ is a basic clause of size at most $k+1$, and hence it follows from Lemma 5.10 that $\sim\widehat{\theta} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$.

In other words:

$$\mathrm{product}(\{(y{:}v_1 \vee \ldots \vee y{:}v_p) \mid y \in \mathrm{parents}(x), Dom(y) = \{v_1, \ldots, v_p\}\}) \subseteq \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$$

It then follows from Proposition 5.4 that $\mathrm{Tr}(C)$ is not $(k+1)$-consistent. ■

5.4.5 Functional Constraints and Intricacy

We now consider functional networks, which include the networks in the tractable class of [DH91], and show that the intricacy of any inconsistent functional network is at most 1. Recall that a constraint $c(x, y)$ between variables x and y of a network is called functional iff for each value in $c(x)$, there is at most one value in $c(y)$ (and vice versa) that satisfies the constraint.

Definition 5.10 A binary constraint network is called *functional* iff each connected component in the network contains a spanning tree of edges representing functional constraints. ■

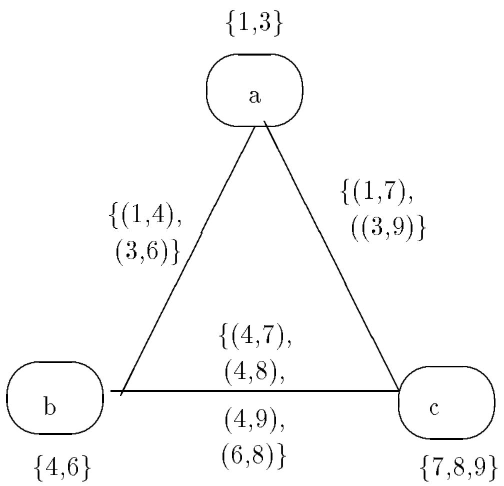

Figure 5.5: A functional constraint network

For example, the network $F1$ given in Figure 5.5 is a functional network, since the edges (a,b) and (a,c) form a spanning tree of functional constraints. The translated theory Tr($F1$) consists of the following formulas:

(Type A)	$(a{:}1 \vee a{:}3)$ $(b{:}4 \vee b{:}6)$ $(c{:}7 \vee c{:}8 \vee c{:}9)$
(Type B)	$(\neg a{:}1 \vee \neg a{:}3)$ $(\neg b{:}4 \vee \neg b{:}6)$
	$(\neg c{:}7 \vee \neg c{:}8)$ $(\neg c{:}7 \vee \neg c{:}9)$ $(\neg c{:}8 \vee \neg c{:}9)$
(Type D)	$((a{:}1 \wedge b{:}4) \vee (a{:}3 \wedge b{:}6))$
	$((a{:}1 \wedge c{:}7) \vee (a{:}3 \wedge c{:}9))$
	$((b{:}4 \wedge c{:}7) \vee (b{:}4 \wedge c{:}8) \vee (b{:}4 \wedge d{:}9) \vee (b{:}6 \wedge d{:}8))$

If $b{:}6$ is added to Tr($F1$), then FP will simplify the Type B formula for b to $\neg b{:}4$. The Type D formula on the functional edge (a,b) is then simplified to $a{:}3$, i.e., the functional edge leads to a specific value being assigned to a. Continuing further, the Type D formula on the functional edge (a,c) gets simplified to $c{:}9$. Finally, the Type D formula on the edge (b,c) gets simplified to **f**, leading to inconsistency.

Alternatively, if $b{:}4$ is added to Tr($F1$), the FP simplifies the Type D formulas on the functional edges (a,b) and (a,c) to $a{:}1$ and $c{:}7$, respectively, without leading to an inconsistency. However, if the tuple $(1,7)$ is removed from the constraint on the edge (a,c), then FP will obtain **f** even in this case from the resulting network, which is functional and inconsistent.

Thus, FP seems well-suited for dealing with functional constraints. Lemma 5.12 generalizes this observation: if $c(x,y)$ is functional and x is assigned a value a then FP either leads to inconsistency or assigns a specific value b to y. For a functional network, it then follows using the spanning tree of functional constraints that assigning a value to any node causes FP to either deduce inconsistency or assign values to all the connected nodes. Theorem 5.13 uses this argument to show that the intricacy of any inconsistent functional network is at most 1.

Lemma 5.12 *For any theory* Γ*, if there is a functional constraint between variables* x *and* y *in a constraint network* $C = (X, V, E, c)$ *then for any* $v \in c(x)$ *such that* $\mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v\} \not\Rightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$ *there exists* $w \in c(y)$ *such that* $\mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v, y{:}w\} \not\Rightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$.

Proof: Consider any $v \in c(x)$ such that $\mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v\} \not\Rightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$. Suppose $c(x,y) = \{(v_1, w_1), \ldots, (v_p, w_p)\}$ for some $p > 0$ (such a p exists, otherwise $c(x,y)$ will be empty and $\mathrm{Tr}(C) \Rightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$, a contradiction). Since

$c(x, y)$ is functional, $v_i \neq v_j$ and $w_i \neq w_j$ for any i and any j such that $i \neq j$.

Suppose $v \neq v_i$ for each i. By construction of $\mathrm{Tr}(C)$, it contains the Type D clause $((x : v_1 \wedge y : w_1) \vee \ldots \vee \ldots (x{:}v_p \wedge y{:}w_p))$ and Type B clauses $(\neg x{:}v \vee \neg x{:}v_i)$ for all i such that $v \neq v_i$. Thus, the theory, say Δ, containing just these two kinds of clauses is a subset of $\mathrm{Tr}(C)$. It follows from Proposition 4.8 that $\Delta \cup \{x{:}v\} \Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$, and then from Proposition 4.3 that $\mathrm{Tr}(C) \cup \{x{:}v\} \Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$, which is a contradiction.

Thus, $v = v_m$ for some m in $1, \ldots, p$. Let $w_m = w$.

$$
\begin{array}{lll}
\{\mathbf{f}\} & \not\Leftrightarrow^*_{\mathrm{FP}} & \mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v\} \quad \text{(given)} \\
 & \Leftrightarrow^*_{\mathrm{FP}} & \mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v, (x{:}v \wedge y{:}w)\} \cup \{\neg x{:}v_i \mid i \neq m\} \\
 & & \quad \text{(by Type B and D clauses and Proposition 4.8)} \\
 & \Leftrightarrow^*_{\mathrm{FP}} & \mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v, y{:}w\} \cup \{\neg x{:}v_i \mid i \neq m\} \quad \text{(Property C}\dagger\text{)}
\end{array}
$$

It then follows from Proposition 4.3 that $\mathrm{Tr}(C) \cup \Gamma \cup \{x{:}v, y{:}w\} \not\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$. ∎

If $\mathrm{Tr}(C) \not\vdash_{\mathrm{FP}} \neg x{:}v$, it follows from Lemma 5.7 that the valuation that just maps x to v is a partial solution of C. Lemma 5.12 shows that if there is a functional constraint between x and y, then this partial solution can be extended to another partial solution by also mapping y to some w such that $\mathrm{Tr}(C) \cup \{x{:}v\} \not\vdash_{\mathrm{FP}} \neg y{:}w$. If there is a functional constraint between y and some other node, the partial solution can be similarly further extended. Theorem 5.13 is proved by repeating this process over a tree of functional constraints.

Theorem 5.13 *For any inconsistent functional network D, the intricacy of* $\mathrm{Tr}(D)$ *is at most 1.*

Proof: All we need to show is that $\mathrm{Tr}(D)$ is not 1-consistent, i.e., $\mathbf{f} \in \mathrm{lfp}(T_{\mathrm{Tr}(D),1})$. Since $\mathrm{Tr}(C) \subseteq \mathrm{Tr}(D)$ for each connected component C of D, it follows from Proposition 4.12 that all we need to show is that $\mathbf{f} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),1})$ for some connected component C of D.

Consider any inconsistent connected component $C = (X, V, E, c)$ of D (since D is inconsistent, there must be such a C). It follows from Proposition 5.4 that all we need to show is that there is some $x \in X$ such that $\mathrm{Tr}(C) \vdash_{\mathrm{FP}} \neg x{:}v$ for each $v \in Dom(x)$. We prove that this, in fact, holds for each $x \in X$.

Suppose, by way of contradiction, there is a $x \in X$ and a $v \in Dom(x)$ such that $\mathrm{Tr}(C) \not\vdash_{\mathrm{FP}} \neg x{:}v$, i.e., $\mathrm{Tr}(C) \cup \{x : v\} \not\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$. Since C has a spanning tree of edges representing functional constraints, there is an ordering $x_1, \ldots, x_n$ (where $n \geq 1$) of all nodes in X such that $x_1 = x$ and for each $i \in 2, \ldots, n$ there is a $j_i < i$ for which the edge (x_{j_i}, x_i) represents a functional constraint. For each $i \in 2, \ldots, n$, let $X_i = \{x_1, \ldots, x_i\}$ and suppose there is a valuation θ_i over X_i such that $\mathrm{Tr}(C) \cup \{\widehat{\theta_i}\} \not\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$. Then it follows from Lemma 5.12 (using the functional constraint on the edge (x_{j_i}, x_i)) and property C† that there is an extension θ_{i+1} of θ_i to X_{i+1} such that $\mathrm{Tr}(C) \cup \{\widehat{\theta_{i+1}}\} \not\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$ (note that the valuation θ_i without the conjunct for variable x_j corresponds to the theory Γ of Lemma 5.12). By considering θ_1 to be the valuation that maps x to v, we obtain from repeated use of the above statement that there is a valuation θ over X such that $\mathrm{Tr}(C) \cup \{\widehat{\theta}\} \not\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$, i.e., $\mathrm{Tr}(C) \not\vdash_{\mathrm{FP}} \widehat{\theta}$. It then follows from Lemma 5.7 that θ solves C, which contradicts that C is inconsistent. ∎

Note that the above proof relies on C being a functional network. However, we do not require that some spanning trees of edges representing functional constraints be known.

5.4.6 A New Family of Tractable Classes

Let us combine the intuitions behind the two tractable classes of the previous sections:

1. As we saw in Lemma 5.10, the adaptive-consistency algorithm of [DP87] is "simulated" by our fixpoint construction, with the induced width of a network bounded below by the intricacy of its translation.

2. Once a node is assigned a value, functional constraints allow fact propagation to obtain values that must be assigned to other nodes.

Recall that intricacy corresponds to the size of the longest basic clauses that need to be considered to obtain **f**; by combining the two previous intuitions, we can reduce the size of the needed clauses to be smaller than the induced width. This yields a new family of tractable classes of binary networks, which includes the tractable classes based on induced width and functional constraints. We will construct an example to show that this inclusion is strict.

Consider the functional network of Figure 5.5 augmented by a new node d and new edges connecting d to each of the original nodes a, b, and c. The set of values allowed for d is $\{0, 2, 5\}$, and the set of tuples allowed for the new edges are the cross-products of the values allowed for the incident nodes, except that the following tuples are NOT allowed:

(1,0) is not allowed for (a,d), (4,2) is not allowed for (b,d), and (7,5) is not allowed for (c,d).

Note that parents(d) $= \{a, b, c\}$ and the induced width of the new network is 3. Recall from Section 5.4.4 that *Adaptive-Consistency* considers valuations over at most 3 variables, and, among others, while processing node d, *Adaptive-Consistency* will rule out the valuation that maps a to 1, b to 4, and c to 7, because there is no value of d consistent with this valuation. This corresponds to the basic clause $(\neg a{:}1 \vee \neg b{:}4 \vee \neg c{:}7)$ being in the fixpoint of index 4 which simulates the algorithm. On the other hand, recall from Section 5.4.5 that it is enough to map b to 4 — the other two assignments (a to 1 and c to 7) are then obtained by FP using the formulas in the translation which correspond to the functional edges (b,a) and (b,c). Thus, the shorter basic clause $(\neg b{:}4)$ is itself in the fixpoint with index 2. This shows that even in non-functional networks, function-like constraints may allow FP to rule out valuations by considering possibly fewer variables than the number of parents.

We first generalize functional constraints to the case when the values of all nodes in a set are completely determined by the values of a subset of the nodes called a *seed*. We then define a notion of functional width that is analogous to the notion of induced width, except that only the sizes of the parents' seeds are considered. A class of binary networks with bounded functional width is then shown to have bounded intricacy, and thus tractable. As with bounded induced width, the proof of this claim is also based on the correctness of the *Adaptive-Consistency* algorithm [DP88]. An important difference is that rather than using all the variables of a valuation ruled out by *Adaptive-Consistency* in constructing the constraint clause in the fixpoint, only the variables in its seed are needed. This is possible because any valuation over the variables in a seed of a set of variables is extended by FP to the entire set.

Definition 5.11 For any constraint theory Γ and any sets X and Y of variables in $vars(\Gamma)$ such that $X \subseteq Y$, X is called a <u>*seed*</u> of Y for Γ iff each valuation θ over X has an extension θ' over Y such that $\Gamma/Y \cup \{\widehat{\theta}\} \Leftrightarrow^{*}_{\text{FP}} \Gamma/Y \cup \{\widehat{\theta'}\}$. A <u>*seed function*</u> for Γ is any function that maps each subset of variables in $vars(\Gamma)$ to its seed for Γ. ∎

For example, consider the binary network $C3 = (X, V, E, c)$ given below:

$$\begin{gathered} X = \{p, q, r\} \\ V = \{1, 2, 3, 4, 5, 6, 7\} \\ E = X \times X \\ c(p) = dom(p) = \{1, 2\} \\ c(q) = dom(q) = \{3, 4\} \\ c(r) = dom(r) = \{5, 6, 7\} \\ c(p, q) = \{(1, 3), (1, 4), (2, 4)\} \\ c(p, r) = c(p) \times c(r) - \{(1, 7), (2, 6)\} \\ c(q, r) = c(q) \times c(r) - \{(3, 6), (4, 5)\} \end{gathered}$$

The three consistent valuations over $\{p, q\}$ are the ones allowed by $c(p, q)$. For each of these, there is exactly one value of node r — 5, 6 and 7 respectively. Thus, $\{p, q\}$ is a seed of $\{p, q, r\}$ for the constraint theory

Tr($C3$). Therefore, the function f, which maps every subset of $\{p,q,r\}$ to itself, except $f(\{p,q,r\}) = \{p,q\}$, is a seed function for $C3$. Another seed function just maps every subset of $\{p,q,r\}$ to itself.

Definition 5.12 Given a binary constraint network $C = (X, V, E, c)$ and a seed function f for the theory Tr(C), the *functional width* of C with respect to f is defined identically to the induced width of C, except that parents(x) in the definition is replaced by f(parents(x)). The functional width of C is defined to be the minimum functional width of C with respect to any seed function for Tr(C). ∎

In our simulation of *Adaptive-Consistency* in Section 5.4.4, each valuation ruled out over parents(x) while processing a node x corresponds to a constraint clause (in the fixpoint) that involves only the literals built from the variables in parents(x). Since each such valuation over parents(x) is completely determined by its restriction on any seed of parents(x) using FP, it is possible to have smaller constraint clauses in the fixpoint.

Lemma 5.14, which is the counterpart of Lemma 5.10, shows that if *Adaptive-Consistency* rules out a valuation θ for a binary network C with functional width k then there is a subclause of $\sim\widehat{\theta}$ in the fixpoint lfp($T_{\mathrm{Tr}(C),k+1}$). Note that $\sim\widehat{\theta}$ itself may not be in the fixpoint, since it may not be even in $E(\Gamma, k+1)$. Actually, the subclause in the fixpoint corresponds to the valuation θ restricted to a seed of the variables in $vars(\widehat{\theta})$. Theorem 5.15, which is the counterpart of Theorem 5.11, then shows that if all valuations over a set of nodes in a network C of functional width k are ruled out by *adaptive-consistency*, then Tr(C) has intricacy at most $k+1$.

Lemma 5.14 *Given any binary network C of functional width $k \geq 1$, if* Adaptive-Consistency *rules out a valuation θ then there is a subclause of $\sim\widehat{\theta}$ in* lfp($T_{\mathrm{Tr}(C),k+1}$).

Proof: The proof is based upon that of Lemma 5.10. Let $C = (X, V, E, c)$ be any binary constraint network of functional width k with respect to some ordering $<$ and seed function f for the theory Tr(C); let us abbreviate the theory Tr(C) as Γ, and the theory lfp($T_{\mathrm{Tr}(C),k+1}$) as Δ. We will prove the claim of the lemma by contradiction.

Suppose the claim is false: let θ be the first valuation ruled out by *Adaptive-Consistency* such that no subclause of $\sim\widehat{\theta}$ is in Δ. Let $x \in X$ be the node being processed by *Adaptive-Consistency* while ruling out θ and Γ' be the set of all the constraint clauses $\sim\widehat{\theta''}$ such that the valuation θ'' is ruled out by *Adaptive-consistency* before processing x. Since all constraint clauses in Γ' satisfy the claim of the lemma, we obtain that each clause in Γ' has a subclause in Δ. Let ψ be the constraint clause $\sim\widehat{\theta}$, and let θ^f be the valuation θ restricted to the variables in f(parents(x)).

Suppose we were able to prove that

$$\Gamma/\mathrm{parents}(x) \cup \{\widehat{\theta}\} \Leftrightarrow^*_{\mathrm{FP}} \Gamma/\mathrm{parents}(x) \cup \{\widehat{\theta^f}\} \qquad \text{— } (I)$$

Since *Adaptive-Consistency* rules out valuation θ at node x, $\widehat{\theta}$ extended so it maps x to v is not a partial solution of $\Gamma \cup \Gamma'$ for any $v \in Dom(x)$. That is, $(\Gamma \cup \Gamma')/(\mathrm{parents}(x) \cup \{x\}) \models (\sim\widehat{\theta} \mathbin{\mathring{\vee}} \neg x:v)$. Using Lemma 5.9, we obtain that $(\Gamma \cup \Gamma')/(\mathrm{parents}(x) \cup \{x\}) \vdash_{\mathrm{FP}} (\sim\widehat{\theta} \mathbin{\mathring{\vee}} \neg x{:}v)$, and then from Proposition 4.3 that $\Gamma \cup \Gamma' \vdash_{\mathrm{FP}} (\sim\widehat{\theta} \mathbin{\mathring{\vee}} \neg x{:}v)$, and then from Property G† and modularity that $\Gamma \cup \Delta \vdash_{\mathrm{FP}} (\sim\widehat{\theta} \mathbin{\mathring{\vee}} \neg x{:}v)$. We then obtain from modularity and (I) that $\Gamma \cup \Delta \vdash_{\mathrm{FP}} (\sim\widehat{\theta^f} \mathbin{\mathring{\vee}} \neg x{:}v)$. Since $(\sim\widehat{\theta^f} \mathbin{\mathring{\vee}} \neg x{:}v)$ is in $E(\Gamma, k+1)$, we then obtain from the fixpoint construction that $(\sim\widehat{\theta^f} \mathbin{\mathring{\vee}} \neg x{:}v) \in \Delta$.

As the above holds for each $v \in Dom(x)$, it follows from Proposition 4.8 that $\Delta \vdash_{\mathrm{FP}} \sim\widehat{\theta^f}$. Using the fixpoint argument again, we obtain that $\sim\widehat{\theta^f} \in \Delta$, which is a contradiction because $\sim\widehat{\theta^f}$ is a subclause of $\sim\widehat{\theta} = \psi$.

Therefore, we are left to establish (I). Since f(parents(x)) is a seed of parents(x), we obtain by the definition of seed that there is an extension θ^o of θ^f over parents(x) for which:

$$\Gamma/\mathrm{parents}(x) \cup \{\widehat{\theta^o}\} \Leftrightarrow^*_{\mathrm{FP}} \Gamma/\mathrm{parents}(x) \cup \{\widehat{\theta^f}\} \qquad \text{— } (II).$$

It remains to show that θ is indeed this extension, i.e., $\theta^o = \theta$. We prove this by contradiction.

Suppose $\theta^o \neq \theta$, i.e., there is at least one variable in $parents(x) - f(parents(x))$ which is assigned different values by θ^o and θ. Let $\theta^\star$ denote the restriction of the valuation θ over the variables in $parents(x) - f(parents(x))$. Hence

$$\begin{aligned}
\Gamma/parents(x) \cup \{\widehat{\theta}\} &\Leftrightarrow^*_{\mathrm{FP}} \Gamma/parents(x) \cup \{\widehat{\theta^f}\} \cup \{\widehat{\theta^\star}\} \ (\theta \text{ combines } \theta^f \text{ and } \theta^\star) \\
&\Leftrightarrow^*_{\mathrm{FP}} \Gamma/parents(x) \cup \{\widehat{\theta^o}\} \cup \{\widehat{\theta^\star}\} \ (\text{modularity of FP and } (II)) \\
&\Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\},
\end{aligned}$$

since θ^o and $\theta^\star$ differ on some variable in $parents(x) - f(parents(x))$. It follows that θ is not a partial solution of the theory $\Gamma/parents(x)$, and therefore of the network $C/parents(x)$. Thus, θ must violate some binary constraint, say, $c(y, z)$, in C where $y, z \in \mathrm{parents}(x)$, because it already satisfies each unary constraint in C. Let θ' be θ restricted to the variables in $\{y, z\}$. Since $\sim\widehat{\theta'}$ is of the form $(\neg y{:}a \vee \neg z{:}b)$ for some values a and b, we obtain from the Type B and D formulas in Γ and the properties C†, D†, and E† of FP that $\Gamma \vdash_{\mathrm{FP}} \sim\widehat{\theta'}$. Since $\sim\widehat{\theta'} \in E(\Gamma, k+1)$ (because $k \geq 1$), we then obtain from the fixpoint construction that $\sim\widehat{\theta'} \in \Delta$, which is a contradiction because $\sim\widehat{\theta'}$ is a subclause of $\sim\widehat{\theta} = \psi$. Thus, θ must be a solution of $C/\mathrm{parents}(x)$, a contradiction. ∎

Theorem 5.15 *For any inconsistent binary network C of functional width k, the intricacy of $\mathrm{Tr}(C)$ is at most $k+1$.*

Proof: The proof is based on that of Theorem 5.11. Suppose $C = (X, V, E, c)$ is an inconsistent binary constraint network of functional width k using the seed function f. We will show that $\mathbf{f} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$. Since C is inconsistent, the functional width k is at least 1 (because E is not empty). Since C is inconsistent, there is a variable $x \in X$ such that each valuation θ' over parents(x) was ruled out by *Adaptive-Consistency*. It follows from Lemma 5.14 that there is a subclause ψ of $\sim\widehat{\theta'}$ such that $\psi \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$. Thus, $\mathrm{Viv}(\mathrm{Tr}(C), k+1) \cup \{\widehat{\theta'}\} \Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$ follows from $\mathrm{Viv}(\mathrm{Tr}(C), k+1) = \mathrm{Tr}(C) \cup \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$ and the properties C†, D†, and E† of FP.

Consider any valuation θ over $f(\mathrm{parents}(x))$. Since $f(\mathrm{parents}(x))$ is a seed of parents(x) for $\mathrm{Viv}(\mathrm{Tr}(C), k+1)$, there must be some extension θ' of θ over parents(x) such that $\mathrm{Viv}(\mathrm{Tr}(C), k+1) \cup \{\widehat{\theta'}\} \Leftrightarrow^*_{\mathrm{FP}} \mathrm{Viv}(\mathrm{Tr}(C), k+1) \cup \{\widehat{\theta}\}$. Since $\mathrm{Viv}(\mathrm{Tr}(C), k+1) \cup \{\widehat{\theta'}\} \Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$ for each valuation θ' over parents(x), we obtain that $\mathrm{Viv}(\mathrm{Tr}(C), k+1) \cup \{\widehat{\theta}\} \Leftrightarrow^*_{\mathrm{FP}} \{\mathbf{f}\}$, i.e., $\mathrm{Viv}(\mathrm{Tr}(C), k+1) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Since $\sim\widehat{\theta} \in E(\Gamma, k+1)$, it then follows from the fixpoint construction that $\sim\widehat{\theta} \in \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$. Since this holds for all valuations over $f(\mathrm{parents}(x))$, we obtain that

$$\mathrm{product}(\{(y{:}v_1 \vee \ldots \vee y{:}v_p) \mid y \in f(\mathrm{parents}(x)), Dom(y) = \{v_1, \ldots, v_p\}\}) \subseteq \mathrm{lfp}(T_{\mathrm{Tr}(C),k+1})$$

It then follows from Proposition 5.4 that $\mathrm{Tr}(C)$ is not $(k+1)$-consistent. ∎

It then follows directly from Theorems 5.1 and 5.15 that the consistency problem is tractable for any class of binary constraint networks for which there is a k such that the functional width of any unsatisfiable network in the class is at most k. In other words, classes of binary networks with bounded functional width are tractable.

As in the case of bounded induced width, our proofs were based upon the correctness of the *Adaptive-Consistency* algorithm. However, we do not require that the algorithm be tractable for any class of networks, nor that some seed function or some ordering that realizes the functional width be known.

In any functional network, it can be shown by using an argument similar to that in the proof of Theorem 5.13 that each singleton subset of any non-empty set X of variables is a seed of X (the extension of any consistent valuation over the seed grows similarly along the edges of the spanning tree which represent the functional constraints; an inconsistent valuation can be extended in any way). It follows that the functional width of any functional network is 1. Since induced width of any binary network is at least its functional width, it then follows from Theorem 5.15 that tractable classes based on bounded functional width include those based on bounded induced width and those based on functional constraints. We now present a class of non-functional binary networks with bounded functional width but unbounded induced width. Thus, it will follow from from Theorem 5.15 that this class is tractable; however, this tractability claim does not follow from the results in [DP88] and [DH91].

This class generalizes the example network $C3$ given earlier in this section to more than 3 nodes, and adds similar networks that are inconsistent. For each network with n nodes, where $n > 2$, there will be an ordering $x_1, \ldots, x_n$ of nodes such that for any $i = 2, \ldots, n$, the set $\{x_1, x_2\}$ will be the seed of the set $\{x_1, \ldots, x_i\}$. Thus, the functional width of each network in this class will be at most 2. Since each network will have a constraint between each pair of nodes, the induced width will be $n - 1$ for each network in the class with n nodes. Since no constraints will be functional, none of the networks will be functional.

Consider the class of all networks $C_n = (X_n, V_n, E_n, c_n)$ for each $n \geq 2$ where

$$
\begin{gathered}
X_n = \{x_1, \ldots, x_n\} \\
V_n = \{1, 2, 3, 4, v_1, \ldots, v_n\} \\
E_n = X_n \times X_n \\
c_n(x_1) = dom(x_1) = \{1, 2\} \\
c_n(x_2) = dom(x_2) = \{3, 4\} \\
c_n(x_k) = dom(x_k) = \{v_1, \ldots, v_k\} \text{ for each } k \text{ s.t. } 3 \leq k \leq n \\
c_n(x_1, x_2) = \{(1,3), (1,4), (2,4)\} \\
c_n(x_1, x_k) = \{1, 2\} \times c_n(x_k) - \{(1, v_3), (2, v_2)\} \text{ for each } k \text{ s.t. } 3 \leq k \leq n \\
c_n(x_2, x_k) = \{3, 4\} \times c_n(x_k) - \{(3, v_2), (4, v_1)\} \text{ for each } k \text{ s.t. } 3 \leq k \leq n \\
c_n(x_i, x_j) = c_n(x_i) \times c_n(x_j) - \{(v_1, v_{i+1}), (v_2, v_{i+1}), (v_3, v_{i+1})\} \text{ for each } i \text{ and } j \text{ s.t. } 3 \leq i < j \leq n
\end{gathered}
$$

For each $k = 3, \ldots, n$ in the network C_n, the constraints $c_n(x_1, x_k)$ and $c_n(x_2, x_k)$ together ensure that there is at most one value from the set $\{v_1, v_2, v_3\}$ which is consistent for node x_k with each pair of values in the constraint $c_n(x_1, x_2)$. Further, for any $k > 3$, the constraints $c_n(x_i, x_k)$ for each $i = 3, \ldots, k-1$ ensures that the value v_{i+1} cannot be consistently assigned to node x_k. Thus, for each subset $\{x_1, \ldots, x_k\}$ of nodes, there is exactly one partial solution corresponding to each pair in $c_n(x_1, x_2)$ — assign the values v_1, v_2, and v_3, respectively, to each node x_i for $i = 3, \ldots, k$. It follows that the set $\{x_1, x_2\}$ is a seed in the network C_n for each set $\{x_1, \ldots, x_k\}$ of nodes, where $3 \leq k \leq n$. Hence, the functional width of any network C_n in the class is 2.

The class given above in not interesting in itself, since each network in the class is consistent. It becomes interesting when we add a network (say, D_n) for each $n \geq 2$ which is identical to C_n except that the values v_1, v_2, and v_3 are removed from the domain $c_n(x_n)$ of node x_n, making D_n inconsistent. We can show for each n, using the same arguments as for C_n, that D_n is not functional, has induced width $n - 1$ and has functional width 2. This class, which contains binary networks C_n and D_n for each $n \geq 2$, is the desired tractable class of non-functional networks with bounded functional width but unbounded induced width.

Theorem 5.15 and the above tractable class together show that the new tractable classes, which are based on bounded functional width, strictly subsume those based on bounded induced width and those based on functional constraints.

5.5 Databases with Disjunctive Information

OR-databases extend the relational data model by allowing explicit representation of disjunctive information [IMV94]: tuples in a relation contain either ordinary constants or so-called OR-objects, each of which is associated with a set of constants, called its domain; an Or-object intuitively represents the potential values that may appear in that position. Each OR-database therefore can be thought of as representing many relational databases, obtained by replacing every OR-object by some value in its domain. In [IMV94], as part of the database schema, one can impose some restrictions on the occurrence of OR-objects; for example one can restrict attention to the repeated occurrences or the columns of the relations in which they can appear.

A query is said to be true in an OR-database iff it is true in all the possible relational databases associated with it. Although querying OR-databases is intractable in general, [IMV94] identifies maximal conditions for tractable querying for several kinds of schemata. These conditions are based on syntactic features of the queries, which can be tested efficiently.

In this section, we first reduce the querying problem to the (un)satisfiability problem in PCE by translating each instance of querying to a theory in PCE. We then show that the translations of each tractable class of querying identified in [IMV94] have bounded intricacy. Our proof for this relies crucially on the results of [IMV94]. Also, since intricacy is only a sufficient condition for tractability, we cannot use it to show that these classes are maximal classes for tractable querying.

It turns out that the precise definitions of the tractable classes of queries given in [IMV94] are not used in our proofs. It suffices for our purposes that the tractable algorithms of [IMV94] are correct for these classes. Since the precise definitions are also technically quite involved, we do not reproduce them here. The interested reader may refer to [IMV94].

5.5.1 Terminology and Notation

In this section, we start by using full first-order predicate calculus in order to describe the problem of querying databases. The following notational conventions will be observed: We use symbols a, b, etc. to denote constants, symbols o, p, etc. to denote OR-objects, symbols u, v, etc. to denote either OR-objects or constants, and symbols x, y, etc. to denote variables. A symbol with an arrow ($\vec{\ }$) over it is used to denote a tuple (i.e., a possibly-empty finite sequence) of objects; for example, $\vec{a}$ denotes a tuple of constants, and $\vec{a}\vec{x}$ (also written as $\vec{a}, \vec{x}$) denotes a tuple of constants followed by a tuple of variables. Upper-case letters are used to denote corresponding sets of objects; for example, O denotes a set of OR-objects.

The following definitions concerning OR-databases are mostly taken from [IMV94]. An OR-database D consists of a domain function Dom together with a finite set of atoms of the form $P(\vec{u})$ where P is some predicate symbol and $\vec{u}$ is a finite sequence of constants or OR-objects.[5] The constants are from a fixed denumerable set $\mathcal{U}$, OR-objects are from another fixed denumerable set $\mathcal{V}$, and the domain function Dom assigns each OR-object o occurring in D a finite non-empty non-singleton set of constants from $\mathcal{U}$. The non-singleton assumption is not restrictive, since any occurrence of an OR-object with singleton domain in a database can be replaced by the unique constant in its domain.

Three kinds of OR-databases are identified, roughly corresponding to the following restrictions: in type III databases, certain columns are restricted to contain no OR-objects; type II databases are type III databases where no OR-object may appear in more than one column; and type I databases further require every OR-object to occur at most once. Variables in a query which correspond to the columns in which OR-objects are allowed are called the OR-arguments of the query.

A mapping θ that assigns to each OR-object o in a database D a constant in $Dom(o)$ is called a *valuation* of D. The result, denoted by $D\theta$, of substituting $\theta(o)$ for each occurrence of an OR-object o in D is known as an *instance* of D.[6] Each such instance is a usual relational database.

[5] In using the tuple notation, we always implicitly assume that the lengths are consistent; for example, the number of symbols in $\vec{u}$ must be equal to the arity of predicate P in the expression $P(\vec{u})$.

[6] These are called *models* in [IMV94].

Similarly, $\vec{u}\theta$ denotes the result of substituting $\theta(o)$ for each occurrence of an OR-object o in the tuple $\vec{u}$ of constants and OR-objects. It follows that

$$D\theta = \{P(\vec{u}\theta) \mid P(\vec{u}) \in D\}$$

A *proper query* is an existentially quantified conjunctive formula $\Phi(\vec{x})$ whose atoms are built from the predicate symbols in D, constants in $\mathcal{U}$, and variables from a fixed denumerable set $\mathcal{W}$, with the restriction that no two literals in the query have the same predicate symbol; the query is said to be *closed* if $\vec{x}$, the free variables in the query, is empty. A query is *normal* iff each literal has at most one OR-argument. We often abbreviate the query $\Phi(\vec{x})$ by Φ.

A query with more that one literal is often written recursively as follows:

$$\Phi(\vec{x}) = \exists \vec{y} P(\vec{x}, \vec{y}) \wedge \Phi_1(\vec{x}, \vec{y})$$

where $P(\vec{x}, \vec{y})$ represents the first literal[7] in which all the variables are from the tuple $\vec{x}\vec{y}$ and $\Phi_1(\vec{x}, \vec{y})$ represents the rest of the query. If we want to be more careful about the variables shared by P and Φ_1, we would express such queries as:

$$\Phi(\vec{x}_1, \vec{x}_2, \vec{x}_3) = \exists \vec{y}_1, \vec{y}_2 P(\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2) \wedge \Phi_1(\vec{x}_2, \vec{x}_3, \vec{y}_2)$$

An OR-database D *entails* a closed query Φ, denoted by $D \approx\!\!\!\mid \Phi$, iff every instance of D satisfies Φ.[8] The *data complexity* of a closed query Φ with respect to a schema S is the complexity of determining whether any given database D that conforms to S belongs to the set $\{D \mid D \approx\!\!\!\mid \Phi\}$; Φ is *tractable* with respect to S iff its data complexity with respect to S is in PTIME.

The *answer set*, $\Phi(D)$ of a query $\Phi(\vec{x})$ to a database D is defined as the set

$$\Phi(D) = \{\vec{a} \mid D \approx\!\!\!\mid \Phi(\vec{a})\}$$

where $\vec{a}$ denotes a tuple of constants.

We denote valuations by conjunctive formulas. For example, the valuation that assigns a_1 to o_1 and a_2 to o_2 is denoted by the formula $(o_1 \doteq a_1 \wedge o_2 \doteq a_2)$. The empty valuation is denoted by the formula $\mathbf{t}$. The set of all possible valuations of D can be conveniently represented as a product of the set defined using the construct given below:

Definition 5.13 For any finite set O of OR-Objects, the set $\mathrm{Or}(O)$ is defined as:

$$\mathrm{Or}(O) = \{(o \doteq a_1 \vee \ldots \vee o \doteq a_n) \mid o \in O \text{ and } Dom(o) = \{a_1, \ldots, a_n\}\}$$

■

The valuations over a set O of OR-objects correspond to the clauses in the set product($\mathrm{Or}(O)$) in the following way: for any finite set O of OR-objects, a formula θ is a valuation over O iff $\sim\theta \in$ product($\mathrm{Or}(O)$). We have to be careful about the extreme case: if O is empty then the only valuation is $\mathbf{t}$ and product($\mathrm{Or}(O)$) = product($\{\}$) = $\mathbf{f}$.

Since a valuation θ of a database D is a conjunction of equality atoms, it follows from property F† that $D \cup \{\theta\} \Leftrightarrow^*_{\mathrm{FPE}} D\theta \cup \{\theta\}$ for any database D and any valuation θ.

5.5.2 Query Answering by Checking Satisfiability

We use the standard technique for mapping the problem of database query-answering to propositional satisfiability: first expressing the problem using entailment in first-order logic, and then transforming it to propositional satisfiability by considering ground instances of the quantified formulas. We need an appropriate domain for generating the ground instances:

[7] [IMV94] denotes such a literal by $l_P(\vec{x}, \vec{y})$.

[8] This relation is denoted by $\models$ in [IMV94]. "Satisfies" here should be taken in the relational database sense: the database yields a model for the query formula.

P	
1	α
2	β
3	α

R	
1	1
2	1
1	2

$Dom(\alpha) = Dom(\beta) = \{1, 2\}$

$\Phi = \exists x \exists y (P(x, y) \wedge R(y, x))$

Figure 5.6: OR-database D_0 and query Φ

$$\begin{array}{lll} P(1,\alpha) & R(1,1) & \alpha = 1 \vee \alpha = 2 \\ P(2,\beta) & R(2,1) & \beta = 1 \vee \beta = 2 \\ P(3,\alpha) & R(1,2) & \end{array}$$

$\neg P(I, J) \vee \neg R(I, J)$ for every combination of values $I, J \in \{1, 2, 3\}$

Figure 5.7: $\mathrm{Tr}(D_0, \Phi, ())$

Definition 5.14 The *domain* $Dom(D)$ of the database D is defined as the set of all constants that either occur in D or are in $Dom(o)$ for some OR-object o that occurs in D. ∎

Transforming entailment to satisfiability requires considering the complement of the query. Since any ground query Φ, i.e., without any variables, is a formula in PCE, its complement $\sim \Phi$ is also a formula in PCE. Recall that $\sim \Phi$ is the formula in negation normal form which is obtained by appropriately pushing the negation in to the literals. For an arbitrary query, the set of ground instances are defined as follows:

Definition 5.15 For any database D and any query Φ, the set $G(\Phi)$ is defined inductively as follows:

1. if Φ has no quantifiers then $G(\Phi) = \{\sim \Phi\}$;
2. if $\Phi = \exists x\ \sigma(x)$ then

$$G(\Phi) = \cup\{G(\sigma(a)) \mid a \in Dom(D)\}$$

∎

It follows that for any database D and any query Φ, the set $G(\Phi)$ consists of ground clausal formulas, and thus is a theory in PCE. Given a query $Q(\vec{x})$ posed to a database D, for every tuple $\vec{a}$ we will use G to construct a theory based on $Q(\vec{a})$ and D, such that the theory is satisfiable iff $\vec{a}$ is *not* in the answer set. The collection of all such theories for a fixed schema and a fixed query provides a class of satisfiability problems.

Definition 5.16 For any database D, any query $\Phi(\vec{x})$, and any tuple $\vec{a}$ of constants, the *translation*, $\mathrm{Tr}(D, \Phi, \vec{a})$, is defined to be the theory

$$D \cup \mathrm{Or}(D_{or}) \cup G(\Phi(\vec{a}))$$

The *translation*, $\mathrm{Tr}(S, \Phi)$, of a query $\Phi(\vec{x})$ with respect to a schema S is defined to be the set

$$\{\mathrm{Tr}(D, \Phi, \vec{a}) \mid D \text{ conforms to } S \text{ and } \vec{a} \text{ is a tuple of constants}\}$$

∎

Since each component of $\mathrm{Tr}(D, \Phi, \vec{a})$ is a theory in PCE, $\mathrm{Tr}(D, \Phi, \vec{a})$ is also a theory in PCE. For an example, consider the OR-database D_0 of Figure 5.6, where both the OR-objects α and β have the same domain $\{1, 2\}$, and the query $\Phi = \exists x \exists y (P(x, y) \wedge R(y, x))$. The translation $\mathrm{Tr}(D_0, \Phi, ())$ contains exactly the formulas given in Figure 5.7, where the last schema represents all formulas obtained by substituting each of I and J by a value in $\{1, 2, 3\}$ in all possible ways. In this example, D_0 entails the query Φ and the theory $\mathrm{Tr}(D_0, \Phi, ())$ is unsatisfiable. More generally, it can be seen that a tuple $\vec{a}$ of constants is an answer to query $\Phi(\vec{x})$ over database D iff $\mathrm{Tr}(D, \Phi, \vec{a})$ is unsatisfiable.

5.5.3 Tractable Algorithms for Querying

We now review the tractable algorithms presented in [IMV94]. Our proofs will be based on the correctness of these algorithms.

The recursive procedure EntailsI[9] is used for querying a Type I database D. EntailsI recursively evaluates a tractable proper query, $\Phi(\vec{x})$, by evaluating the subquery obtained by removing the first literal.

Procedure EntailsI($\Phi(\vec{x}), D$)

```
/* D is a Type I database;
Φ(x⃗) = ∃y⃗P(x⃗, y⃗) ∧ Φ₁(x⃗, y⃗);
Returns the answer set Φ(D) */
 1. A := EntailsI(Φ₁(x⃗, y⃗), D);
 2. return({a⃗ | exists P(u⃗) ∈ D such that
      for every valuation θ of u⃗, there is a b⃗ for which
      P(u⃗θ) = P(a⃗, b⃗) and ⟨a⃗, b⃗⟩ ∈ A})
```

end (EntailsI).

In the base case, when $\Phi(\vec{x}) = \exists\vec{y}P(\vec{x},\vec{y})$ and Φ_1 is empty, A is considered to be the set of all tuples. The same example which is given after the algorithm for type II databases (next) illustrates EntailsI.

The recursive procedure EntailsII is used for querying a Type II database D. EntailsII decomposes a tractable normal proper query, $\Phi(\vec{x})$, in two different ways, depending on how the variables are shared among the literals of the query. In principle, it is similar to EntailsI except that more than one tuple of the relation corresponding to the first literal is combined with the answer to the recursive calls; this is because different tuples may share OR-objects. If the query contains no OR-arguments, then a standard query evaluation method is used.

Procedure EntailsII($\Phi(\vec{x}), D$)

```
/* D is a Type II database;
Φ(x⃗) contains some OR-argument;
Returns the answer set Φ(D) */
 1. if Φ(x⃗₁, x⃗₂) = ∃y⃗z[P(x⃗₁, y⃗, z) ∧ Φ₁(x⃗₁, x⃗₂, y⃗) ∧ Φ₂(z)] then {
 2.    A₁ := EntailsII(Φ₁(x⃗₁, x⃗₂, y⃗), D);
 3.    A₂ := EntailsII(Φ₂(z), D);
 4.    return({⟨a⃗₁, a⃗₂⟩ | exists P(a⃗₁, b⃗, o) ∈ D such that ⟨a⃗₁, a⃗₂, b⟩ ∈ A₁ and
          oθ ∈ A₂ for each valuation θ})
 5. }
 6. /* Φ(x⃗) is of the form Φ₁(x⃗) ∧ ∃y⃗z[P(y⃗, z) ∧ Φ₂(y⃗, z)] */
 7. A₁ := EntailsII(Φ₁(x⃗, D);
 8. A₂ := EntailsII(Φ₂(y⃗, z), D);
 9. if there exists o such that for each valuation θ
      there exists constants b⃗ for which
      P(b⃗, o) ∈ D and ⟨b⃗, d⟩ ∈ A₂ then
10.       return(A₁)
11.       else return(∅)
```

end (EntailsII).

Consider the type II database D_0 and query Φ of Figure 5.6. EntailsII decomposes the query with empty ψ_1 and $\psi_2 = R(y,x)$. Thus, the set A_1 is not relevant, and the set $A_2 = R$ (with arguments reversed). If $o = \alpha$, then for each d in $Dom(o) = \{1,2\}$, there is a tuple $\langle 1, \alpha\rangle$ in P and constant $b = 1$ such that tuple

[9] The exact names of the algorithms and their exact steps differ from those in [IMV94], since they use the same name for more than one algorithm, and the notation is somewhat different. The essential ideas, however, remain the same.

$\langle 1, d\rangle$ is in A_2. Thus, the query is answered "yes". Note that, even if we had removed the tuple $\langle 3, \alpha\rangle$ from P to make D_0 a type I database, EntailsI would return "yes".

The recursive procedure EntailsIII, used for querying a Type III database D, is quite different from EntailsI and EntailsII. Although EntailsIII decomposes a tractable normal proper query, $\Phi(\vec{x})$, as in Entails1, it attempts to remove those values which cause a "yes" answer from the domains of OR-objects. It iterates this process, until either there is a definite "yes" answer, or no such reduction in domain is possible. This iteration is necessary, since OR-objects can be shared across columns of the database.

Procedure EntailsIII(Φ, D)

/* D is a Type III database;
$\Phi = \exists\vec{x}P(\vec{x}) \wedge \Phi_1(\vec{x})$;
Returns ''yes'' iff $\Phi(D) \neq \emptyset$ */
1. V := set of OR-objects in P relation of D;
2. D^0 := D; $Dom^0 = Dom$; i := 0;
3. L^0 := $\{\vec{a} \mid$ EntailsIII$(\Phi_1(\vec{a}), D)\}$;
4. repeat {
5.   if there is o in V such that for each valuation θ
      there is $\vec{a}$ for which $P(\vec{a}) \in D^i\theta$ and $\vec{a} \in L^i$
      then return(''yes'');
6.   for each $o \in V$, modify the domains as follows:
      $Dom^{i+1}(o)$:= $Dom^i(o) - \{d \mid$ there is $\vec{a}$ for which
          $P(\vec{a}) \in D^i(o \doteq d)$ and $\vec{a} \in L^i$ $\}$;
7.   let D^{i+1} be the database obtained from D^i
      by replacing Dom^i by Dom^{i+1};
8. L^{i+1} := $\{\vec{a} \mid$ EntailsIII$(\Phi_1(\vec{a}), D^{i+1})\}$;
9.   if $L^{i+1} = L^i$ then return(''no'')
10.   else i := i+1
11. }
end (EntailsIII).

Since the conditions in lines 5 and 6 are almost identical, it is possible to combine them. This is done by removing line 5, and adding the following line after line 6:

return ''yes'' if $Dom^{i+1}(o) = \emptyset$ for some o in V;

It follows directly that the modified algorithm, say EntailsIII$'$, returns the same answers as EntailsIII. EntailsIII$'$ turns out to be more useful for our purposes. Intuitively, each iteration of the "repeat" loop removes those values from the domain of OR-objects which will cause the query to be answered "yes". The algorithm returns "yes" iff the domain of some OR-object becomes empty in this process.

5.5.4 Describing Tractable Queries Using Intricacy

We will show that the translations of the tractable cases identified in [IMV94] produce classes of theories with bounded intricacy. We will prove our claim for each of the three types of databases by induction on the length of the queries. Our proofs will be based on the correctness of the algorithms presented in [IMV94], which are reviewed in Section 5.5.3, for answering the tractable queries. The basic idea is to show that any tuple is in the answer set for a database iff the translated theory is not k-consistent, where k is a fixed number determined by the query. Since the translated theory is unsatisfiable iff the tuple is in the answer set, we will obtain the desired result, i.e., the intricacy of each unsatisfiable theory is at most k.

Our proofs of non k-consistency will essentially "simulate" the algorithms of [IMV94], in the sense that queries will be recursively decomposed in the same way as in the algorithms. There are two intermediate lemmas. Lemma 5.16 will be used for proving the base cases of induction, where the query is a single literal.

Lemma 5.17 will be a technical result about the translated theories, which will be used for extending the inductive hypothesis to the full query containing an additional literal. This technical result will be used in the inductive cases of our proofs.

We now illustrate the intuitions behind our proof for Type II databases using the database D_0 and the query Φ given in Figure 5.6. Recall from Section 5.5.3 that $\psi_2 = R(y,x)$, $A_2 = R$ (with arguments reversed), and the OR-object contributing to the "yes" answer is α, whose domain is $\{1,2\}$. Suppose Γ denotes the translated theory $\mathrm{Tr}(D_0, \Phi, ())$. Consider the two valuations of α: θ_1 which assigns 1, and θ_2 which assigns 2. The two relevant tuples, corresponding to these valuations, in A_2 are $\langle 1,1\rangle$ and $\langle 1,2\rangle$, respectively.

For the base cases, we show that the translated theory for the query ψ_2 has intricacy 0, for each of the two valuations (and the corresponding tuple in A_2) of α. Recall that a theory has intricacy 0 if the rewrite system FPE reduces it to $\{\mathbf{f}\}$. Consider the valuation θ_1 and the tuple $\langle 1,1\rangle$ in A_2. The translated theory for this case is

$$\Gamma_1 = Tr(R\theta_1, R(y,x), \langle 1,1\rangle) = \{R(1,1), R(2,1), R(1,2), \neg R(1,1)\}$$

Using Property D† of FPE, we obtain that Γ_1 is equivalent (with respect to FPE) to the theory

$$\{\mathbf{f}, R(2,1), R(1,2), \neg R(1,1)\}$$

which is equivalent (with respect to FPE) to $\{\mathbf{f}\}$, using Proposition 4.2. Thus, Γ_1 is not 0-consistent. Similarly, $\Gamma_1 = \mathrm{Tr}(R\theta_2, R(y,x), \langle 1,2\rangle)$ is also not 0-consistent.

Now consider the inductive case. We will first show that both $\neg(\alpha = 1)$ and $\neg(\alpha = 2)$ are in the fixpoint $\mathrm{lfp}(T_{\Gamma,1})$. Consider the theory $\Gamma \cup \{\alpha = 1\}$, some of whose formulas are $P(1,1)$, $R(1,1)$, and $\neg P(1,1) \vee \neg R(1,1)$. Using property D†, this subset of formulas is equivalent (with respect to FPE) to $\{P(1,1), R(1,1), \mathbf{f} \vee \mathbf{f}\}$, which is then equivalent (with respect to FPE) to $\{\mathbf{f}\}$. It follows that $\neg(\alpha = 1)$ is in $\mathrm{lfp}(T_{\Gamma,1})$. We can similarly show that $\neg(\alpha = 2)$ is in $\mathrm{lfp}(T_{\Gamma,1})$

Thus, some of the formulas in $\Gamma \cup \mathrm{lfp}(T_{\Gamma,1})$ are $\neg(\alpha = 1)$, $\neg(\alpha = 2)$, and $(\alpha = 1) \vee (\alpha = 2)$. Using property D† as above, we obtain that $\Gamma \cup \mathrm{lfp}(T_{\Gamma,1})$ is equivalent (with respect to FPE) to $\{\mathbf{f}\}$. Thus, Γ is not 1-consistent.

The base case of the above argument is formalized in Lemma 5.16. Lemma 5.17 formalizes the argument used in obtaining that $\neg(\alpha = 1)$ and $\neg(\alpha = 2)$ are in the fixpoint $\mathrm{lfp}(T_{\Gamma,1})$. Lemma 5.19 formalizes the argument used in obtaining that Γ is not 1-consistent. The proofs of both Lemmas 5.16 and 5.19 depend on Proposition 5.4 about products, given in Section 5.3.

Lemma 5.16 *For any database D, any query $\Phi(\vec{x}) = \exists \vec{y} P(\vec{x}, \vec{y})$, any tuple $\vec{a}$ of constants, and any tuple $\vec{u}$ of constants and OR-objects: if $P(\vec{a}, \vec{u}) \in D$, identical variables in $\vec{x}\vec{y}$ correspond to identical entries in $\vec{a}\vec{u}$ (i.e., $\vec{x}, \vec{y}$ unifies with $\vec{a}\vec{u}$), and k is the number of OR-objects in $\vec{u}$ then $\mathrm{Tr}(D, \Phi, \vec{a})$ is not k-consistent.*

Proof: Abbreviate the theory $\mathrm{Tr}(D, \Phi, \vec{a})$ by Γ. Given the conditions of the lemma, we need to show that Γ is not k-consistent. We show this by a case analysis on k:

Suppose $k = 0$, i.e., $\vec{a}\vec{u}$ is a sequence of constants. Since $\neg P(\vec{a}, \vec{u}) \in G(\Phi(\vec{a}))$, we have $\{P(\vec{a}, \vec{u}), \neg P(\vec{a}, \vec{u})\} \subseteq \Gamma$. From property D† using Propositions 4.2 and 4.3, we obtain that $\Gamma \vdash_{\text{FPE}} \mathbf{f}$, i.e., Γ is not 0-consistent.

Now consider the case when $k > 0$. Suppose $O = \{o_1, \ldots, o_k\}$ is the set of OR-objects in $\vec{u}$. Consider any $\varphi \in \mathrm{product}(\mathrm{Or}(O))$ and let θ be $\sim\varphi$. It follows that θ is a valuation over O, $P(\vec{a}, \vec{u}\theta) \in D\theta$, and $\neg P(\vec{a}, \vec{u}\theta) \in G(\Phi(\vec{a}))$ since $\vec{a}, \vec{u}\theta$ is a sequence of constants. Thus, using the same reasoning as above, we obtain that $\Gamma \cup \{D\theta\}$ is not 0-consistent, and then from Proposition 4.12 that $\Gamma \cup \{D\theta\} \cup \{\theta\}$ is not 0-consistent. Since $D \subseteq \Gamma$ and $D \cup \{\theta\} \Leftrightarrow^*_{\text{FPE}} D\theta \cup \{\theta\}$, it follows from Proposition 4.12 that $\Gamma \cup \{\theta\}$ is not 0-consistent, and then from Proposition 5.5 that $\sim\theta = \varphi \in \mathrm{lfp}(T_{\Gamma,k})$. Since this is true for any $\varphi \in \mathrm{product}(\mathrm{Or}(O))$, we obtain that $\mathrm{product}(\mathrm{Or}(O)) \subseteq \mathrm{lfp}(T_{\Gamma,k})$. It then follows from Proposition 5.4 that Γ is not k-consistent. ■

Lemma 5.17 *For any database D, any proper query $\Phi(\vec{x}_1, \vec{x}_2, \vec{x}_3) = \exists \vec{y}_1 \vec{y}_2 P(\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2) \wedge \Phi_1(\vec{x}_2, \vec{x}_3, \vec{y}_2)$, any tuples $\vec{a}_1, \vec{a}_2, \vec{a}_3$ of constants, any tuples $\vec{u}_1, \vec{u}_2$ of constants and OR-objects, any valuation θ over the OR-objects in $\vec{u}_1\vec{u}_2$, and any number k: if $P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \in D\theta$ and $\mathrm{Tr}(D\theta, \Phi_1, \vec{a}_2\vec{a}_3\vec{u}_2\theta)$ is not k-consistent then $\sim\theta \in \mathrm{lfp}(T_{\Gamma,k+m})$, where m is the number of OR-objects in $\vec{u}_1\vec{u}_2$ and Γ is the theory $\mathrm{Tr}(D, \Phi, \vec{a}_1\vec{a}_2\vec{a}_3)$.*

Proof: Denote the theory $\text{Tr}(D\theta, \Phi_1, \vec{a}_2\vec{a}_3\vec{u}_2\theta)$ by Δ, and the theory $D \cup \text{Or}(D_{or}) \cup G(P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \wedge \Phi_1(\vec{a}_2, \vec{a}_3, \vec{u}_2\theta))$ by Γ'. Note that $\Gamma' \subseteq \Gamma$.

$$\begin{array}{rcl}
\Delta \cup \{\theta\} & = & \{\theta\} \cup D\theta \cup \text{Or}(D_{or}) \cup G(\Phi_1(\vec{a}_2, \vec{a}_3, \vec{u}_2\theta)) \\
& & \quad \text{(definition)} \\
& \Leftrightarrow^*_{\text{FPE}} & \{\theta\} \cup D\theta \cup \text{Or}(D_{or}) \cup G(P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \wedge \Phi_1(\vec{a}_2, \vec{a}_3, \vec{u}_2\theta)) \\
& & \quad \text{(property E}\dagger\text{ and modularity, since } P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \in D\theta) \\
& \Leftrightarrow^*_{\text{FPE}} & \{\theta\} \cup \Gamma' \quad \text{(using modularity, since } \{\theta\} \cup D\theta \Leftrightarrow^*_{\text{FPE}} \{\theta\} \cup D)
\end{array}$$

Since Δ is not k-consistent, it follows from Proposition 4.12 that $\Delta \cup \{\theta\}$ is not k-consistent, and then from Proposition 4.12 that $\{\theta\} \cup \Gamma'$ is not k-consistent. Since $\Gamma' \subseteq \Gamma$, it follows from Proposition 4.12 that $\{\theta\} \cup \Gamma$ is not k-consistent, and then from Proposition 5.5 that $\sim\theta \in \text{lfp}(T_{\Gamma, k+m})$. ∎

The next three propositions prove our claim for type I, II, and III databases, respectively. As remarked earlier, the precise definitions of the tractable classes of queries given in [IMV94] are not used in our proofs. It suffices for our purposes that the tractable algorithms of [IMV94] are correct for these classes.

Proposition 5.18 *For any type I database D, any tractable proper query $\Phi(\vec{x})$, and any tuple $\vec{a}$ of constants: $\vec{a} \in \Phi(D)$ iff $\text{Tr}(D, \Phi, \vec{a})$ is not k-consistent, where k is the number of OR-arguments in the query.*

Proof: If $\text{Tr}(D, \Phi, \vec{a})$ is not k-consistent, then $\mathbf{f} \in \text{lfp}(T_{\text{Tr}(D,\Phi,\vec{a}),k})$. It then follows from Theorem 4.13 that $\text{Tr}(D, \Phi, \vec{a})$ is not satisfiable, i.e., $\vec{a} \in \Phi(D)$.

We will prove the *only-if* side of the claim by induction on the number of literals (say, n) in the query Φ:

(base case, $n = 1$): The query $\Phi(\vec{x})$ is of the form $\exists\vec{y}P(\vec{x}, \vec{y})$. Suppose $\vec{a} \in \Phi(D)$. We obtain from EntailsI that there is a tuple $\vec{u}$ of constants and OR-objects such that $P(\vec{a}\vec{u}) \in D$, and $\vec{x}, \vec{y}$ unifies with $\vec{a}\vec{u}$. Using Lemma 5.16, we obtain $\text{Tr}(D, \Phi, \vec{a})$ is not p-consistent, where p is the number of OR-objects in $\vec{u}$. Since $p \leq k$, we obtain from Proposition 4.12 that $\text{Tr}(D, \Phi, \vec{a})$ is not k-consistent.

(inductive case, $n > 1$): It follows from Lemma 4.1 of [IMV94] that the query Φ is of the form $\Phi(\vec{x}_1, \vec{x}_2, \vec{x}_3) = \exists\vec{y}_1, \vec{y}_2 P(\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2) \wedge \Phi_1(\vec{x}_2, \vec{x}_3, \vec{y}_2)$, where the inductive conditions hold for the subquery Φ_1. Suppose $\vec{a} = \vec{a}_1\vec{a}_2\vec{a}_3 \in \Phi(D)$. We obtain from EntailsI of [IMV94] that there is a tuple $\vec{u}_1\vec{u}_2$ of constants and OR-objects such that for any valuation θ of OR-objects in $\vec{u}_1\vec{u}_2$, we have $P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \in D\theta$ and $\vec{a}_2\vec{a}_3\vec{u}_2\theta \in \Phi_1(D)$.[10] Let O denote the set of OR-objects in $\vec{u}_1\vec{u}_2$.

Since Φ_1 has fewer than n literals, we can use the inductive hypothesis to obtain that $\text{Tr}(D, \Phi_1, \vec{a}_2\vec{a}_3\vec{u}_2\theta)$ is not q-consistent, where q is the number of OR-arguments in Φ_1. Using Proposition 4.12, we obtain $\text{Tr}(D\theta, \Phi_1, \vec{a}_2\vec{a}_3\vec{u}_2\theta)$ is not q-consistent. Using Lemma 5.17, we then obtain that $\sim\theta \in \text{lfp}(T_{\Gamma,p+q})$ where $\Gamma = \text{Tr}(D, \Phi, \vec{a})$ and p is the number of OR-objects in $\vec{u}_1\vec{u}_2$.

Since the above holds for each valuation θ, we obtain that $\text{product}(\text{Or}(O)) \subseteq \text{lfp}(T_{\Gamma,p+q})$. Using Proposition 5.4, we obtain that Γ is not $(p+q)$-consistent. Since $q \leq (k-p)$, we obtain from Proposition 4.12 that Γ is not k-consistent. ∎

Proposition 5.19 *For any type II database D, any tractable normal proper query $\Phi(\vec{x})$, and any tuple $\vec{a}$ of constants: $\vec{a} \in \Phi(D)$ iff $\text{Tr}(D, \Phi, \vec{a})$ is not k-consistent, where k is the number of OR-arguments in the query.*

Proof: The proof of Proposition 5.18 will work for this proposition as well, though it will use EntailsII instead of EntailsI and Lemma 5.4 instead of Lemma 4.1 of [IMV94]. In the base case, when the query $\Phi(\vec{x})$ is

[10] Though EntailsI uses a stronger condition, in which each $P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta)$ should come from the same tuple in the database D, the weaker version suffices for our proof. Also, this weaker version is needed for the proof regarding Type *II* databases.

of the form $\exists \vec{y} P(\vec{x}, \vec{y})$ and $\vec{a} \in \Phi(D)$, it again follows (now from EntailsII) that there is a tuple $\vec{u}$ of constants and OR-objects such that $P(\vec{a}\vec{u}) \in D$, and $\vec{x}, \vec{y}$ unifies with $\vec{a}\vec{u}$. Note that both ways of decomposing the query in EntailsII are covered by the same inductive case of that proof; we had used a weaker condition in the inductive case in anticipation of this theorem. Also, since each literal in a normal query contains at most one OR-argument, p is at most 1 in both the base and the inductive cases. ∎

We now prove the claim for type *III* databases, based on the correctness of the algorithm EntailsIII′ (the modified version of the algorithm given in Figure 5 of [IMV94]) of Section 5.5.3. The basic idea in our proof is to show that each value removed from a domain forces the corresponding literal in the fixpoint.

Proposition 5.20 *For any type III database D, any tractable normal proper query $\Phi(\vec{x})$, and any tuple $\vec{a}$ of constants: $\vec{a} \in \Phi(D)$ iff $\mathrm{Tr}(D, \Phi, \vec{a})$ is not k-consistent, where k is the number of OR-arguments in the query.*

Proof: If $\mathrm{Tr}(D, \Phi, \vec{a})$ is not k-consistent, then $\mathbf{f} \in \mathrm{lfp}(T_{\mathrm{Tr}(D,\Phi,\vec{a}),k})$. It then follows from Theorem 4.13 that $\mathrm{Tr}(D, \Phi, \vec{a})$ is not satisfiable, i.e., $\vec{a} \in \Phi(D)$.

We will prove the *only-if* side of the claim by induction on the number of literals (say, n) in the query. Let's use Γ to denote the theory $\mathrm{Tr}(D, \Phi, \vec{a})$ and Δ to denote the theory $\mathrm{Viv}(\Gamma, k)$.

The base case ($n = 0$) is exactly as in the proof of Proposition 5.18. Suppose the query $\Phi(\vec{x})$ is of the form $\exists \vec{y} P(\vec{x}, \vec{y})$ and $\vec{a} \in \Phi(D)$. We obtain from EntailsIII′ that there is a tuple $\vec{u}$ of constants and OR-objects such that $P(\vec{a}\vec{u}) \in D$, and $\vec{x}, \vec{y}$ unifies with $\vec{a}\vec{u}$.

For the inductive case ($n > 0$), it follows from Lemma 5.4 [IMV94] that the query $\Phi(\vec{x})$ is of the form $\Phi(\vec{x}_1, \vec{x}_2, \vec{x}_3) = \exists \vec{y}_1, \vec{y}_2 P(\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2) \wedge \Phi_1(\vec{x}_2, \vec{x}_3, \vec{y}_2)$, where the inductive conditions hold for the subquery Φ_1. Suppose $\vec{a} = \vec{a}_1\vec{a}_2\vec{a}_3 \in \Phi(D)$. Thus, for some $r > 0$, the modified algorithm returns "yes" answer in the rth iteration of the "repeat" loop. Thus, for some OR-object, say o, $Dom^r(o) = \emptyset$ in D^r. We will show by induction on i that for any $i \in 0 \ldots r$, any OR-object w in D:

$$\forall d : d \in Dom(w) - Dom^i(w) \Rightarrow (w \not\doteq d) \in \mathrm{lfp}(T_{\Gamma,k})$$

If we succeed in showing this, then, since this should hold for $w = o$, we obtain $\forall d \in Dom(o) : (o \not\doteq d) \in \mathrm{lfp}(T_{\Gamma,k})$. In other words, $\mathrm{product}(\mathrm{Or}(\{u\})) \subseteq \mathrm{lfp}(T_{\Gamma,k})$. It then follows from Proposition 5.4 that Γ is not k-consistent.

All that is left is to prove the inductive claim mentioned above. The base case, when $i = 0$, is trivial since $Dom(w) = Dom^0(w)$. So consider the inductive case, when $i > 0$. If $Dom^i(w) = Dom^{i-1}(w)$ then the result follows from the inductive hypothesis. Otherwise, consider any $d \in Dom^{i-1}(w) - Dom^i(w)$ and let $\theta = (w \doteq d)$. We obtain from EntailsIII′ that there is a tuple $\vec{u}_1\vec{u}_2$ of constants and OR-objects such that $P(\vec{a}_1, \vec{a}_2, \vec{u}_1\theta, \vec{u}_2\theta) \in D^{i-1}\theta$ and $\vec{a}_2\vec{a}_3\vec{u}_2\theta \in \Phi_1(D^{i-1}\theta)$. From the outer inductive hypothesis, we obtain that $\mathrm{Tr}(D^{i-1}\theta, \Phi_1, \vec{a}_2\vec{a}_3\vec{u}_2\theta)$ is not p-consistent, where p is the number of OR-arguments in Φ_1. Using Lemma 5.17, we obtain that $(w \not\doteq d) \in \mathrm{lfp}(T_{\Gamma^{i-1}}, p+1)$ where Γ^{i-1} denotes the theory $\mathrm{Tr}(D^{i-1}, \Phi, \vec{a})$. Since $p + 1 \leq k$, we obtain from Proposition 4.12 that $(w \not\doteq d) \in \mathrm{lfp}(T_{\Gamma^{i-1},k})$. Using the inner inductive hypothesis, we obtain from modularity and property E† that $(w \not\doteq d) \in \mathrm{lfp}(T_{\Delta',k})$, where $\Delta' = \Gamma \cup \{(w \not\doteq d) \mid d \in Dom(w) - Dom^{i-1}(w)\}$. Thus, $(w \not\doteq d) \in \mathrm{lfp}(T_{\Gamma,k})$. This proves the inner inductive claim. ∎

It follows directly from the above three propositions that $\mathrm{Tr}(\Phi, S)$ for any closed proper query Φ and any OR-database schema S has bounded intricacy in each of the following three cases:

1. Φ is tractable with respect to the type I schema S,
2. the normal query Φ is tractable with respect to the type II schema S, and
3. the normal query Φ is tractable with respect to the type III schema S.

The bound on the intricacy in each case is the number of OR-arguments in the query Φ.

Since the syntactic criteria of [IMV94] provide a complete characterization of tractable queries, it follows that bounded intricacy also provides a complete characterization of these tractable queries: a proper closed query Φ is tractable with respect to a schema S (Φ is also required to be normal if S is of type II or III) iff there is a $k \in \mathcal{N}$ such that the intricacy of each unsatisfiable theory in $\mathrm{Tr}(\Phi, S)$ is at most k.

Note that our proofs rely crucially on the results in [IMV94]. Moreover, the characterization of tractable queries in [IMV94] is syntactic and can be checked very efficiently, while determining intricacy of any given theory is not known to be an easy problem. However, we have shown that our notion of "bounded intricacy" does cover exactly all the tractable closed proper queries for the OR-databases considered in [IMV94].

5.6 Disjunctive Logic Programming

For our purposes, a disjunctive logic program (DLP) [LJR92] is a basic clausal theory in PC, without the equality predicate $\doteq$. A class of DLP is tractable iff the satisfiability problem for it is tractable. From our previous results, we know that the family of DLP with intricacy k is tractable, but we have no efficient syntactic check to determine the intricacy of any particular DLP.

We present here a new family $\mathrm{DLP}(1), \mathrm{DLP}(2), \ldots$ of tractable classes of DLP for which membership can be tested in polynomial time, and which have the property that all unsatisfiable DLP in class $\mathrm{DLP}(k)$ have intricacy at most k.

In effect, the construction of class $\mathrm{DLP}(k)$ is inspired by the notion of intricacy: To determine that the intricacy of a theory is less than k, we need to consider, according to the definition, all clauses of size k or less, which is an exponential number. How could we limit the set of clauses that need to be looked at? One idea is to consider only those clauses (of size k or less) which are *immediate subclauses of existing clauses* in the theory. By also fixing an ordering on the clauses, we ensure that the number of clauses considered is only polynomial in both k and the size of the DLP. To make things work, while constructing a fixpoint, we simplify the DLP by removing any clause with a subclause in the fixpoint and by keeping the theory irreducible with respect to FP.

Our main result is then that if the class of simplified DLPs is known to have bounded intricacy, then the original DLPs also have bounded intricacy. Thus, we obtain a family of tractable DLPs classes, one class for each value of k. Since the rewrite system FPE is identical to FP for PCE without equality, we use FP in this section.

5.6.1 A Tractable Family of DLPs

We present the details of the family $\mathrm{DLP}(1), \mathrm{DLP}(2), \ldots$ of DLPs and prove that they are tractable. We also present a polynomial time algorithm for testing membership in each of them. The classes are defined inductively: the base class $\mathrm{DLP}(1)$ consists of theories with intricacy 1 that are discussed in Section 2.8. For each $k \geq 2$, the class $\mathrm{DLP}(k)$ consists of theories where considering the immediate subclauses of only the clauses of size at most k in the theory simplifies it to a theory in $\mathrm{DLP}(1)$.

Let $\ggcurly$ be any total ordering on all basic clauses of PC without equality. $\mathrm{DLP}(1)$, the base class, is defined to the class containing all Horn, 2-CNF, positive, and negative theories. For any $k > 1$, the class $\mathrm{DLP}(k)$ is defined using the algorithm Member-dlp given in Figure 5.8.

For any DLP Γ and number $k \geq 2$, the algorithm Member-dlp iterates over each non-unit clause of Γ of size at most k. It then iterates over each immediate subclause of the selected clause. If the selected subclause is inferred using $\vdash_{\mathrm{FP}}$ from Γ then the subclause replaces the selected clause in Γ. This process continues until either Γ is in $\mathrm{DLP}(1)$ or no such subclause can be inferred. Note that in line 4 of the algorithm, σ is the next clause in the ordering $\ggcurly$, some of whose immediate subclauses have not been selected (in line 5) since the last change to the theory Γ. In line 5, μ is the next immediate subclause of σ in the ordering $\ggcurly$ which has not been selected since the last change to the theory Γ. The repeat loop terminates when there is no change to Γ for all possible choices of σ and μ.

For example, consider $k = 3$ and theory $\Gamma_1 = \{(P \vee Q \vee R), (P \vee Q \vee \sim R)\}$. For $\sigma = (P \vee Q \vee R)$ and

Algorithm Member-dlp:

Input: `a DLP` Γ`, a number` $k \geq 2$`;`
Output: `''yes'' if` $\Gamma \in \mathrm{DLP}(k)$`; ''no'' otherwise;`
1. `repeat`
2. $\quad\Gamma := \mathrm{FPF}(\Gamma)$;
3. $\quad$`if` $\Gamma \in \mathrm{DLP}(1)$ `then return ''yes'';`
4. $\quad$`select next clause` $\sigma \in \Gamma$ `where` $2 \leq |\sigma| \leq k$`;`
5. $\quad$`select next immediate subclause` μ `of` σ`;`
6. $\quad$`if` $\Gamma \vdash_{\mathrm{FP}} \mu$ `then`
7. $\qquad\Gamma := \Gamma - \{\sigma\} \cup \{\mu\}$;
8. `until no change in` Γ`;`
9. `return ''no'';`

end.

Figure 5.8: Testing membership in DLP(k)

$\mu = (P \vee Q)$, we have $\mathrm{FPF}(\Gamma_1 \cup \{\sim\mu\}) = \{\mathbf{f}\}$. Thus, the new theory, say Γ_1', is $\{(P \vee Q), (P \vee Q \vee \sim R)\}$. Now, for $\sigma = (P \vee Q \vee \sim R)$ and $\mu = (P \vee Q)$, we have $\mathrm{FPF}(\Gamma_1' \cup \{\sim\mu\}) = \{\mathbf{f}\}$. Since the new theory $\{(P \vee Q), (P \vee Q)\}$ is a 2-CNF theory, $\Gamma_1 \in \mathrm{DLP}(3)$.

For another example, consider $k = 3$ and theory $\Gamma_2 = \{(\sim P \vee Q \vee R), (\sim P \vee Q \vee \sim S), (\sim P \vee \sim R \vee S)\}$. For $\sigma = (\sim P \vee Q \vee R)$ and $\mu = (\sim P \vee Q)$, we have $\mathrm{FPF}(\Gamma_2 \cup \{\sim\mu\}) = \{\mathbf{f}\}$. Since the new theory $\{(\sim P \vee Q), (\sim P \vee Q \vee \sim S), (\sim P \vee \sim R \vee S)\}$ is a Horn theory, $\Gamma_2 \in \mathrm{DLP}(3)$.

Lemma 5.21 *The intricacy of any unsatisfiable clausal DLP in class* DLP(k) *is at most* k.

Proof: Consider any unsatisfiable clausal DLP, Π, in the class DLP(k). All we need to show is that Π is not k-consistent. For $k = 1$, it follows from the observations in Section 2.8 that Π is not 1-consistent. Now we consider the case when $k > 1$.

Let Δ be a variable whose value at any step of the algorithm Member-dlp (with input Π and k) is the theory obtained from Π by performing all additions and deletions of clauses that happened at line 7 until that step of the execution (i.e., ignoring all calls to FPF). It follows from the modularity and convergence of FP that the assertion $\mathrm{FPF}(\Delta) = \Gamma$ is an invariant just after line 2. Since algorithm Member-dlp returns "yes" and any unsatisfiable DLP in DLP(1) is not 1-consistent, FPF(Δ) is also not 1-consistent for the last value of Δ, just before termination.

We now show that if a clause μ is added to Γ at line 7, then $\mu \in \mathrm{lfp}(T_{\Pi,k-1})$. Note that any such μ is in $E(\Pi, k-1)$. Assume the claim is false — let μ be the first clause that violates this claim and let Γ' and Δ' be the values of Γ and Δ just before μ is added to Γ; then $\Delta' \subseteq \Pi \cup \mathrm{lfp}(T_{\Pi,k-1})$. From line 6 we have that $\Gamma' \vdash_{\mathrm{FP}} \mu$, and thence, from the invariant and the modularity of FP, that $\Delta' \vdash_{\mathrm{FP}} \mu$. It then follows from Proposition 4.3 that $\Pi \cup \mathrm{lfp}(T_{\Pi,k-1}) \vdash_{\mathrm{FP}} \mu$, and then from the definition of $\mathrm{lfp}(T_{\Pi,k-1})$ that $\mu \in \mathrm{lfp}(T_{\Pi,k-1})$. This contradiction proves our claim.

Thus, $\Delta \subseteq \Pi \cup \mathrm{lfp}(T_{\Pi,k-1})$ every time through the loop. Since at the end FPF(Δ) is not 1-consistent and $k > 1$, it follows from Proposition 4.12 that $\Pi \cup \mathrm{lfp}(T_{\Pi,k-1})$ is not $(k-1)$-consistent. Using the fixpoint construction, we then obtain that Π is not $(k-1)$-consistent. ∎

It follows from the above lemma and Theorem 5.1 that DLP(k) is tractable for each k. We now show that even a naive implementation of algorithm Member-dlp runs in time $O(n^3)$, where n is the size of Γ. For this, we will require that the total ordering $\ggg$ on clauses be efficiently computable, for example, a lexicographic ordering based on some total ordering on literals.

Lemma 5.22 *If $\ggcurly$ is a lexicographic ordering then for any DLP Γ of size n, algorithm Member-dlp with input Γ takes time $O(n^3)$.*

Proof: We assume that the atoms in each clause as well as the clauses in Γ are always kept sorted. This is easy to enforce at the start by a preprocessing step that does the sorting: this can be done in $O(n \; log \; n)$ time. Later, we always add the new clause μ in line 7 in such a way that the sorting is preserved: this can be done in $O(n)$ time. With this ordering, the two selections in lines 4 and 5 can be made by properly maintaining counters to record the previous selections: this can be done in $O(log \; n)$ time each.

Since each clause in Γ can be shortened to a unit clause, line 7 can be executed at most $O(n)$ times. Since each addition takes $O(n)$ time, the total cost of executing line 7 is $O(n^2)$. We now compute the cost of executing other lines.

The clause and the subclause that causes the change each time could be the last possible selections, causing $O(n)$ iterations of the repeat loop each time line 7 is executed. The cost of each iteration is dominated by the calls to FPF in lines 2 and 7. Since Γ is a clausal theory, each of these calls cost $O(n)$ time. Thus, the total time for executing the algorithm Member-dlp is $O(n^3)$. ∎

We next sketch how the time complexity of determining membership in the class DLP(k) can be further reduced to quadratic in the size of the input DLP. The idea is to exploit the modularity of FPF to always incrementally compute the results of the two calls to FPF in lines 2 and 7. The incremental version of the algorithm explicitly maintains $\Delta(\sigma, \mu) = \mathrm{FPF}(\Gamma \cup \{\sim\mu\})$ for all immediate subclauses μ of all clauses σ in Γ. Each time a clause μ is replaced by an immediate subclause σ which differs only in some literal α, the literal α is removed from the clause obtained from μ in each of the Δs so maintained; FPF is then incrementally applied to each of the new Δs. The incremental algorithm also has to create new Δs corresponding to the immediate subclauses of σ that are added to Γ.

As argued in the proof of the above lemma, the total cost of preprocessing (which does the sorting) and executing line 7 is $O(n^2)$. The incremental algorithm starts with n different Δs. Each time a clause of size m is replaced by a immediate subclause, $m-1$ new Δs are created. Since each clause can be shortened to a unit clause in the worst case, at most $n(k-1)/2$ new Δs are created. Thus, the total number of Δs ever created is $n(k+1)/2$. Since the running time of the algorithm is dominated by executing FPF on each of these Δs, costing $O(n)$ time each, the total execution time is $O(n^2k)$. Thus, for any fixed k, the incremental algorithm runs is quadratic time.

5.7 Conclusions

We presented the bounded intricacy criterion for tractability based on a fixpoint construction using the rewrite system FP: given any class of theories in PCE and some fixed number k, if each unsatisfiable theory in the class has intricacy less than k then the satisfiability problem for the class is tractable. This criterion seems potentially valuable, since several previously identified non-trivial tractable classes have bounded intricacy. (The criterion is not trivial as demonstrated by the class of theories encoding the pigeon hole principle, which is tractable but does not bounded intricacy.) This criterion was useful for identifying new non-trivial tractable classes, as demonstrated in the areas of constraint satisfaction and disjunctive logic programming. We have not yet found any new non-trivial tractable class based on another criterion for tractability which requires that the intricacy of *satisfiable* theories in the class be bounded.

Note that the bounded intricacy criterion does not require theories to be in conjunctive normal form, thus allowing us to use it for constraint satisfaction problems, for example.

Since we have no efficient algorithm to determine the intricacy of (unsatisfiable) theories, we propose to use the bounded intricacy criterion only as a theoretical tool, not to be used at run-time, to analyze classes of theories in PCE. Apart from discovering new tractable classes of satisfiability, it can be used for another closely related task: given a class of theories, determine whether the satisfiability problem for this class is tractable. All we need to do is to analyze the fixpoint construction for unsatisfiable theories for detecting the generation of the clause **f**. This obviates the need for discovering new algorithms for this class and proving the correctness and tractability of at least one of them. Note that our approach is guaranteed to sometimes fail, since bounded intricacy is not a necessary condition for tractability. Also, even if our approach works, it does not necessarily provide the most efficient algorithm for the given class of theories which has been found to be tractable. Note that the various properties of least fixpoints and admissible rewrite systems are useful in proving results about intricacy.

Chapter 6

Conclusions

The goal of this research was to deal with the intractability of reasoning in KR systems. We made contributions to three out of the four approaches listed in Chapter 1 for dealing with the intractability problem. In this chapter, we review our contributions, discuss the limitations of our approach, and suggest directions for future research

6.1 Contributions

Boolean Constraint Propagation (BCP) [McA80, McA90] is widely used for linear-time but incomplete reasoning with clausal propositional theories. However, none of its extensions to the non-clausal case, that have been proposed previously, are known to be tractable (i.e., provably in PTIME). We developed fact propagation (FP), which tractably extends BCP to non-clausal theories. We presented a quadratic-time algorithm for FP, which runs in linear time for clausal theories. Moreover, FP is proved to be more complete than CNF-BCP, a previously-proposed extension of BCP to non-clausal theories. We know of no other reasoner for arbitrary propositional theories that is tractable and at least as complete as FP.

There is a considerable interest in developing anytime reasoners [BD88], which are complete reasoners that provide partial answers even if they are stopped prematurely; the completeness of the answer improves with the time used in computing the answer. Anytime reasoners could be also used for providing a quick "first-cut" to a problem, which can be improved later. Extending FP, we developed a family of increasingly-complete tractable reasoners which could be used for specifying the partial answers of an anytime reasoner. Although families of increasingly-complete tractable reasoners were previously known for the clausal case (c.f. [GS88, CS92a]), we know of no other such family of reasoners for arbitrary propositional theories. Our technique for generating these reasoners is based on restricting the length of the clauses used in chaining (i.e., Modus Ponens). We provided an alternative characterization, based on a fixpoint construction using FP, of the reasoners in our anytime family. This fixpoint characterization was then used to define a transformation of arbitrary propositional theories into logically equivalent "vivid" theories, i.e., theories for which the tractable reasoner FP is complete (our definition of vividness).

Since reasoning problems in particular applications are often restricted cases of general reasoning, it is important to identify tractable classes of reasoning problems. Based on FP, we developed a new property, called bounded intricacy, which is shared by a variety of tractable classes that were presented previously, for example, in the areas of propositional satisfiability (clausal), constraint satisfaction, and OR-databases. Although proving bounded intricacy for these classes required domain-specific techniques, which are based on the original tractability proofs, bounded intricacy is one more tool available for showing that a family of problems arising in some application is tractable. As we demonstrated in the case of constraint satisfaction and disjunctive logic programs, bounded intricacy (for low values of intricacy) can be also used to uncover new tractable classes, which can then be checked for applicability. Since there are tractable classes that do not have bounded intricacy, bounded intricacy also seems to provide some new insights into the structure of tractable problems. Filtering out classes with unbounded intricacy may be used as a "first cut" in eliminating intractable classes of reasoning problems.

6.2 Directions for Future Research

We now suggest several directions for future research, which are mainly motivated by the limitations of our current approach. Success in the directions for future research might remove some limitations.

Rewrite Systems for Tractable Reasoning: We have used a rewrite system to specify FP, a sound and tractable (but incomplete) reasoner for propositional logic. We believe that rewrite systems (with restrictions such as convergence, etc.) can be used more generally, as a tool for describing (tractable) reasoners. To support this hypothesis, numerous other issues need to be first resolved: Can we use rewrite systems for specifying tractable reasoners for other logics, for example, first-order logic and intuitionistic logic? Will the properties of convergence and modularity be useful for such rewrite systems? Would we need more such properties for restricting the space of rewrite systems that are useful for tractable reasoning? Can we axiomatize in a traditional way the logical consequence relations based on these rewrite systems? Can we provide model-theoretic semantics for them? Positive answers to these questions would be very useful.

FP is an admissible rewrite system for incomplete reasoning. However, it seems possible to add some more rewrite rules to FP, and yet retain admissibility. It seems unlikely that there are "maximally-admissible" rewrite systems, in the sense that adding any new rule makes them inadmissible. It is more likely that there are a number of admissible rewrite systems that are incomparable in completeness, possibly with different time complexities. For any task, ideally we should be able to select one or more of these depending on the requirements of the specific task. It would be useful to develop a systematic approach for generating and selecting among these admissible rewrite systems, possibly in a task-specific way. A possible approach might be in the style of the automated completion algorithm of Knuth and Bendix [KB70] which systematically adds rewrite rules for obtaining confluent rewrite systems.

Efficient Vivification and Reasoning: We introduced a transformation (Viv) on theories that makes them vivid, i.e., converts them into logically equivalent theories for which an efficient refutation-reasoner based on FP is complete in inferring clauses. However, Viv is defined using a fixpoint construction which is very inefficient. In particular, there are many clauses that could be ignored during the construction of the fixpoint, and yet the same vivid theory would be obtained. A direction for future research is to develop an optimized version of Viv transformation that adds as few clauses as possible for obtaining a vivid theory. This would increase the efficiency of the various reasoners in the family presented in Chapter 4, since they are characterized based on the Viv transformation. Of course, a lower bound complexity result on the time needed to compute Viv would be useful to indicate the limits of such improvements. A related research issue is to develop a model-theoretic semantics for these reasoners.

Proving Bounded Intricacy: Although we used the notion of product (Section 5.3) several times in proving the bounded intricacy of some class of theories, we needed to rely on several occasions on previous, domain-specific proofs in order to prove that some classes of theories have bounded intricacy. It would be useful to develop a more powerful and general proof technique for demonstrating bounded intricacy — one that can be tried on any new class of theories that is encountered. This would advance the art of developing tractable deduction algorithms for classes of theories that are found to be of interest.

All our tractability claims are based on bounding the intricacy of unsatisfiable theories in a class. We know that bounding the intricacy of satisfiable theories also leads to tractability. Further research is needed to determine whether this criterion leads to any new interesting tractability classes.

Experimental Evaluation and Applications: The algorithms for rewriting with respect to FP and for vivification have not been implemented. We need to implement them and experiment with reasoning problems in order to determine their efficacy. These problems could be randomly generated or may be obtained from some "real-world" application. Recently, there have been some interesting empirical results regarding the difficulty of solving randomly-generated reasoning problems (c.f. see [CKT91, MSL92]). It would be useful to determine whether similar results hold for our algorithms.

Approximations: This work contributes indirectly towards the fourth approach (mentioned in Section 1.1) to the intractability of reasoning, namely, explicitly approximating the information told to the KR system, and/or the queries asked of it. In related work [DE92] (based on earlier work in [BE89]), we restrict the *internal* representation to a tractable subset of the highly expressive Ask and Tell languages; information and queries in the expressive languages are then suitably approximated by the formulas in the tractable language. An open problem is to apply intricacy to find tractable languages appropriate for these approximations.

To illustrate this approach, consider a KR system that maintains positive propositional information about people and their occupations. Even if we restrict to information about a single person, it can be shown that answering queries is intractable, essentially because indeterminate information, in the form of disjunctions, is hard to reason with. Suppose however that we select a (small) subset of disjunctions which we choose to represent accurately — this set is called the *vocabulary* of approximations. Now the KR system, when told some sentence either represents it precisely, or if not possible, approximates it, in a principled way, using some formula in the vocabulary. For example, the statement "Mary is either a lecturer or a teacher" may be approximated to "Mary is an educator" where "educator" is a disjunction of "lecturer", "teacher", and "professor". (This of course may lead to some loss of information.)

The KR system containing these approximations can then be used for answering queries posed to the KR system. Some queries can be answered precisely, given what is stored in the KR system. For example, the query "Is Mary an educator?" will be answered correctly. In other situations, answering the query is itself too hard; in this case the query is also approximated. For example, while both the queries, "Is Mary a teacher or a professor?" and "Is Mary a lecturer or a teacher?" are answered "yes" after approximating them, this answer is correct only for the later query. In many cases, however, we can guarantee either the soundness or completeness of the answers. The main payoff is that computing these approximate answers is often tractable.

6.3 Summary

This document contributes some approaches to deal with the intractability of deductive reasoning in knowledge representation systems. It presents the only known tractable extension of boolean constraint propagation to non-clausal theories. It presents a family of increasingly complete, sound, and tractable reasoners, which can be used for anytime reasoning. It presents a new tool for proving that a family of problems arising in some application is tractable.

Appendix A

Axiomatic Proof Theory for PCE

An axiomatization of PCE is obtained in the usual way by adding the propositional variants of reflexivity and substitutivity axioms for equality [Fit90] to an axiomatization of PC [Men64]. For the purpose of this axiomatization alone, we use the abbreviation $\psi \rightarrow \varphi$ to denote the formula $\vee(\sim\psi, \varphi)$. For any n, if P is any n-place predicate, a, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ are any constants, and ψ, φ, and ω are any formulas of PCE, then the following are axioms of PCE:

1. $a \doteq a$
2. $(a_1 \doteq b_1 \wedge \ldots \wedge a_n \doteq b_n) \rightarrow (P(a_1, \ldots, a_n) \rightarrow P(b_1, \ldots, b_n))$
3. $\psi \rightarrow (\varphi \rightarrow \psi)$
4. $(\psi \rightarrow (\varphi \rightarrow \omega)) \rightarrow ((\psi \rightarrow \varphi) \rightarrow (\psi \rightarrow \omega))$
5. $(\sim\varphi \rightarrow \sim\psi) \rightarrow ((\sim\varphi \rightarrow \psi) \rightarrow \varphi)$

The only rule of inference of PCE is *modus ponens*: φ is a direct consequence of ψ and $\psi \rightarrow \varphi$.

PCE can also be viewed as first-order predicate calculus, FOPC, with equality [Men64] but without:

1. any variables and quantifiers, and
2. any functions other than constants;

except in the axioms (including equality) and the rules of inference.

Appendix B

A Ptime CNF Transformation

Although there is a well-known [Coo71] clausal form transformation that does not cause exponential increase in size of theories, it requires adding new atoms corresponding to their subformulas. We will show that the clausal form of a disjunctive formula is always irreducible with respect to CBCP, that is, clausal BCP does not do any simplification of the clausal form of disjunctive formulas. Since this holds even for disjunctive subformulas of conjunctive formulas, it follows that this clausal form transformation is not at all conducive for reasoning with clausal BCP. We first formally define this transformation:

Definition B.1 For any formula ψ in a theory Γ, the formula $\hat{\psi}$ is defined as follows:

1. if ψ is a fact then $\hat{\psi} = \psi$;
2. otherwise, $\hat{\psi}$ is a new generated atom that does not occur in Γ, and has not been generated before.

The function PCNF, which maps formulas and theories to clausal theories, is defined recursively as follows:

1. for any fact α, $\mathrm{PCNF}(\alpha) = [\![ ]\!]$;
2. for any formula ψ, $\mathrm{PCNF}(\wedge(\psi)) = \mathrm{PCNF}(\vee(\psi)) = \mathrm{PCNF}(\psi)$;
3. for any formulas $\psi = \psi_1 \vee \ldots \vee \psi_n$ $(n \geq 2)$,
   $\mathrm{PCNF}(\psi) = \bigcup\{\mathrm{PCNF}(\psi_i) \mid i = 1 \ldots n\} \cup \left[\!\!\left[\sim\hat{\psi} \vee \hat{\psi_1} \vee \ldots \vee \hat{\psi_n}\right]\!\!\right] \cup \left[\!\!\left[\hat{\psi} \vee \sim\hat{\psi_i} \mid i = 1 \ldots n\right]\!\!\right]$;
4. for any formulas $\psi = \psi_1 \wedge \ldots \wedge \psi_n$ $(n \geq 2)$,
   $\mathrm{PCNF}(\psi) = \bigcup\{\mathrm{PCNF}(\psi_i) \mid i = 1 \ldots n\} \cup \left[\!\!\left[\hat{\psi} \vee \sim\hat{\psi_1} \vee \ldots \vee \sim\hat{\psi_n}\right]\!\!\right] \cup \left[\!\!\left[\sim\hat{\psi} \vee \hat{\psi_i} \mid i = 1 \ldots n\right]\!\!\right]$;
5. for any bag B of formulas, $\mathrm{PCNF}(\odot(B)) = \bigcup\{\mathrm{PCNF}(\psi) \mid \psi \in B\} \cup \left[\!\!\left[\hat{\psi} \mid \psi \in B\right]\!\!\right]$.

∎

For example, consider the theory $\Gamma = \odot((Q \vee (P \wedge (\neg P \vee Q))))$ and suppose the new atoms generated are: A for $(\neg P \vee Q)$, D for $(P \wedge A)$, and C for $(Q \vee D)$. The Ptime CNF transformation, $\mathrm{PCNF}(\Gamma)$ is the theory, say Δ, consisting of the following formulas:

$$
\begin{array}{rcc}
\neg A \vee \neg P \vee Q, & A \vee P, & A \vee \neg Q, \\
D \vee \neg P \vee \neg A, & \neg D \vee P, & \neg D \vee A, \\
\neg C \vee Q \vee D, & C \vee \neg Q, & C \vee \neg D, \\
C & &
\end{array}
$$

Since Δ is irreducible with respect to CBCP, clausal BCP fails to obtain Q from the clausal form of Γ. In general, since the clausal form of a disjunctive formula does not contain any unit clause, the clausal form is irreducible with respect to CBCP. Thus, this transformation is not at all conducive for reasoning with clausal BCP.

Appendix C

Local Consistency in CSP

Approximate methods to determine all solutions of a constraint network strengthen the constraints to obtain an equivalent network such that some local consistency criteria are satisfied [Fre82]. We show that this processing of networks is closely related to our notion of vivification of constraint theories, as defined in Section 4.4.

Definition C.1 [from [Fre82]] For any $k \in \mathcal{N}$, a constraint network C is said to be *k-consistent* iff for any set of $k-1$ variables along with values for each that satisfy all the constraints among them, there exists a value for any kth variable such that the the k values together satisfy all constraints among the k variables. C is said to be *strong k-consistent* iff it is j-consistent for each $j \leq k$. ∎

We now define a parameterized notion of local consistency for constraint theories.

Definition C.2 A *local consistency criterion*, Φ, is a boolean function over the cross-product of the set of constraint theories and the set of valuations, such that $\Phi(\Gamma, \theta)$ for any complete valuation θ over any constraint theory Γ.

For any $k \in \mathcal{N}$, a constraint theory Γ is *k-consistent with respect to a local consistency criterion* Φ iff for any subset X of $vars(\Gamma)$ containing $k-1$ variables, any variable $x \in vars(\Gamma) - X$, any valuation θ over X, if $\Phi(\Gamma, \theta)$ then there is a valuation θ' over $X \cup \{x\}$ such that θ' is an extension of θ and $\Phi(\Gamma, \theta')$. A theory is *strong k-consistent* with respect to Φ iff it is j-consistent with respect to Φ for each $j \leq k$. ∎

Notice that we are dealing with three (different but related) notions of k-consistency in this section. First is k-consistency for constraint networks as defined in [Fre82], second is k-consistency for logical theories, as defined in Section 5.3, and the third is k-consistency of a constraint theory with respect to a local consistency criterion, as defined above. We will be careful in avoiding any mix-up among them: the first applies to networks, while the others to theories; the third will always be associated with some consistency criterion.

We will consider two specific local consistency criteria that are defined as follows:

- $\Phi_S(\Gamma, \theta)$ is true iff θ is a partial solution of Γ;
- $\Phi_D(\Gamma, \theta)$ is true iff $\Gamma \not\vdash_{\text{FP}} \sim\widehat{\theta}$.

It follows directly from the definitions that any constraint network C is k-consistent iff the translated theory $\text{Tr}(C)$ is k-consistent with respect to Φ_S. Thus, techniques in the literature to achieve higher levels of local consistency of constraint networks [Dec91] also achieve higher levels of local consistency of the translated theory with respect to Φ_S. We will show that, in a similar way, our technique of vivification achieves higher levels of local consistency of the translated theory with respect to Φ_D. For this, we need to restrict to a special class of constraint theories.

Definition C.3 A constraint theory Γ is *propagatable* iff $\Gamma \vdash_{\mathrm{FP}} \sim\theta$ for any valuation θ that is not a partial solution of Γ. ∎

Since $\vdash_{\mathrm{FP}}$ is sound, the consistency criterion Φ_D is at least as strong as the criterion Φ_S for propagatable theories, i.e., given any propagatable theory Γ and any valuation θ, if $\Phi_D(\Gamma, \theta)$ then $\Phi_S(\Gamma, \theta)$. Lemma C.1 shows that constraint theories obtained by translations of constraint networks are propagatable. Lemma C.2 shows that making a propagatable theory vivid keeps it propagatable. Theorem C.3 relates vividness and strong k-consistency for propagatable theories.

Lemma C.1 *For any constraint network C, $\mathrm{Tr}(C)$ is propagatable.*

Proof: First we will show that for any constraint network C and any complete valuation θ that does not solve $\mathrm{Tr}(C)$: $\mathrm{Tr}(C) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. then we will extend this to any valuation, possibly not complete.

Consider any network $C = (X, V, E, c)$ and any complete valuation θ that does not solve $\mathrm{Tr}(C)$. We obtain from Lemma 5.7 that θ does not solve C. Since θ is a valuation, $\theta(x) \in Dom(x) = c(x)$ for each $x \in X$, i.e., θ satisfies all unary constraints in c. Thus, some binary constraint, say $c(x, y)$, is violated by θ. Using Proposition 4.8, $\mathrm{Tr}(C)/\{x, y\} \cup \{\widehat{\theta}\} \Leftrightarrow^*_{\mathrm{FPE}} \{\mathbf{f}\}$, i.e., $\mathrm{Tr}(C)/\{x, y\} \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Using Proposition 4.3, we obtain that $\mathrm{Tr}(C) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$.

Now suppose θ be a valuation over X which is not a partial solution of $\mathrm{Tr}(C)$. Since θ does not solve $\mathrm{Tr}(C)/X$, it follows from Lemma 5.8 that θ does not solve $\mathrm{Tr}(C/X)$, and then from the above result that $\mathrm{Tr}(C/X) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Using Lemma 5.8 again, the observation $\mathrm{Tr}(C)/X \subseteq \mathrm{Tr}(C)$, and Proposition 4.3, we obtain that $\mathrm{Tr}(C) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$. Thus, $\mathrm{Tr}(C)$ is propagatable. ∎

Lemma C.2 *For any theory Γ and any number k, if Γ is propagatable then $\mathrm{Viv}(FP, \Gamma, k)$ is propagatable.*

Proof: Suppose Γ is any propagatable theory and k is any number:

Valuation θ is not a partial solution of $\mathrm{Viv}(FP, \Gamma, k)$

- $\Rightarrow$ $\mathrm{Viv}(FP, \Gamma, k) \cup \{\widehat{\theta}\}$ is not satisfiable
- $\Rightarrow$ $\Gamma \cup \{\widehat{\theta}\}$ is not satisfiable (using Theorem 4.13)
- $\Rightarrow$ θ is not a partial solution of Γ
- $\Rightarrow$ $\Gamma \vdash_{\mathrm{FP}} \sim\widehat{\theta}$ (since Γ is propagatable)
- $\Rightarrow$ $\mathrm{Viv}(FP, \Gamma, k) \vdash_{\mathrm{FP}} \sim\widehat{\theta}$ (using Proposition 4.3)

Thus, $\mathrm{Viv}(FP, \Gamma, k)$ is propagatable. ∎

Theorem C.3 *For any propagatable theory Γ, if Γ is k-consistent then Γ is strong k-consistent with respect to Φ_D.*

Proof: Suppose Γ is not strong k-consistent with respect to Φ_D, i.e., there is a $p \leq k$ such that Γ is not p-consistent with respect to Φ_D. Thus there is a valuation θ over $p-1$ variables in $vars(\Gamma)$ and a variable $x \in vars(\Gamma) - vars(\theta)$ such that $\Phi_D(\Gamma, \theta)$ and not $\Phi_D(\Gamma, \theta \cup \{x{:}v\})$ for each $v \in Dom(x)$ (because Γ is propagatable). Using Proposition 5.4, where $\Pi = (x{:}v_1 \vee \ldots \vee x{:}v_n)$ and $Dom(x) = \{v_1, \ldots, v_n\}$, we obtain that $\Gamma \cup \{\widehat{\theta}\}$ is not 1-consistent. We then obtain from Proposition 5.5 that $\sim\widehat{\theta} \in \mathrm{lfp}(T_{\Gamma,p})$. Since $p \leq k$ and Γ is k-vivid, we obtain that $\Gamma \vdash_{\mathrm{FP}} \sim\widehat{\theta}$, i.e., not $\Phi_D(\Gamma, \theta)$, which is a contradiction. ∎

For any satisfiable constraint network C and any number k, it follows from Lemmas C.1 and C.2 that $\mathrm{Viv}(FP, \mathrm{Tr}(C), k)$ is propagatable. Since $\mathrm{Viv}(FP, \mathrm{Tr}(C), k)$ is k-consistent, it follows from Theorem C.3 that $\mathrm{Viv}(FP, \mathrm{Tr}(C), k)$ is strong k-consistent with respect to Φ_D. Thus, vivification is a technique for achieving higher levels of local consistency of constraint theories with respect to Φ_D.

Appendix D

Case Analysis for Proving Confluence

We show various cases of overlap that are considered in proving the confluence of the rewrite systems FPC, FPL, FP, and FPE. Table D.1 summarizes the cases that are considered, while the other tables provide the details of the cases.

The rows of Table D.1 indicate the outer rule schemas, while the columns indicate the inner rule schemas. The first row for each outer rule schema considers inner rules of the same kind (for example, both and-rules), while the second row considers inner rules of the opposite kind (for example, and-rules for or-rules). However, the three rows for E2 corresponds to theory-theory, theory-or, and theory-and rules, respectively. The symbol "$*$" indicates variable overlap, "$-$" indicates that the case never arises, "s" indicates that the case is covered by its symmetric counterpart (i.e., by switching inner and outer rules), and "x" indicates that the case is not maningful. Wherever possible, theory connective is used instead of conjunction.

For example, the first row for the outer rule schema S2 shows that only one case is explicitly covered, namely, overlap between $S2_\vee$ and $S3_\vee$. Three other cases (namely, S2 and S1, S2 and L1, S2 and L2) are covered by symmetry, and the other cases of overlaps do not occur.

Details of the overlaps are shown in the rest of the tables. For each case, we list the outer and inner rule schemas (in the first and the second rows, respectively), the substitution which causes the overlap, the overlap term, the two terms of the critical pair, the common term they rewrite to, and the extra rules used in this rewriting. In case of same meta-variables used in both outer and inner rule schemas, suffixes $'$ and $''$, respectively, are added to differentiate them. The symbol $\overline{\alpha_i}$ abbreviates the literals $\alpha_1, \ldots, \alpha_n$. Various cases are indicated by conditions inside the braces { and }. "Extra" denotes the extra rules used in each case. In Tables D.8 to D.14, each row is split into two lines, so as to fit on the page; the two rows of each case are then separated by dotted lines.

For example, the first case in Table D.2 considers the overlap between $S1_\wedge$ (outer rule) and $S2_\wedge$ (inner rule). Since the two rules have the same meta-variable B, it is renamed to B' and B'', respectively. The two rules then are:

$$S1_\wedge : \wedge(\mathbf{f}, B') \ \Rightarrow\ \wedge(\mathbf{f})$$

$$S2_\wedge : \wedge(\mathbf{t}, B') \ \Rightarrow\ \wedge(B'').$$

The substitution $B' = \{\mathbf{t}\} \cup B$ and $B'' = \{\mathbf{f}\} \cup B$ for some B produces the overlap term $\wedge(\mathbf{f}, \mathbf{t}, B)$. This term rewrites to $\wedge(\mathbf{f})$ and $\wedge(\mathbf{f}, B)$, respectively, by the two rules. Since the second term of this critical pair rewrites to $\wedge(\mathbf{f})$ using $S1_\wedge$, the common term is $\wedge(\mathbf{f})$. Note that no rule, other than $S1_\wedge$ and $S2_\wedge$, is used in arriving at this common term.

outer inner	$S1$	$S2$	$S3$	$P1$	$L1$	$L2$	$F1$	$E1$	$E1*$	$E2$
$S1$	$-$ $-$	$\wedge\wedge$ $-$	$-$ $-$	s $-$	s $-$	s $-$	$-$ $-$	$-$ $-$	$-$ $-$	s $-$
$S2$	s $-$	$-$ $-$	$\vee\vee$ $-$	$-$ $-$	s $-$	s $-$	$-$ $-$	$-$ $-$	$-$ $-$	$-$ $-$
$S3$	s $-$	s $-$	$-$ $-$	$-$ $-$	s $-$	s $-$	$-$ $-$	$-$ $-$	$-$ $-$	$-$ $-$
$P1$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	$-$ $-$	$\wedge\wedge$ $-$	s $-$	s $-$	s $-$	$-$ $-$	x x	s $-$
$P1*$	$\wedge\wedge$ $\wedge\vee$	$-$ $-$	$-$ $-$	$\wedge\wedge$ $\wedge\vee$	$\wedge\wedge$ $\wedge\vee$	$\wedge\wedge$ $\wedge\vee$	$\wedge\wedge$ $\wedge\vee$	x x	$\wedge\wedge$ $\wedge\vee$	s s
$L1$	$\wedge\wedge$ $-$	$\vee\vee$ $-$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	$-$ $-$	$\wedge\wedge$ $\vee\wedge$	$-$ $-$
$L2$	$\wedge\wedge$ $-$	$\vee\vee$ $-$	$\wedge\wedge$ $-$	$\wedge\wedge$ $-$	s $-$	$-$ $-$	$-$ $-$	$-$ $-$	$\vee\vee$ $\vee\wedge$	$-$ $-$
$F1$	$-$ $\vee\wedge$	$-$ $\wedge\vee$	$-$ $\wedge\vee$	$\wedge\wedge$ $\wedge\vee$	$-$ $\wedge\vee$	$-$ $\wedge\vee$	$\wedge\wedge$ $\wedge\vee$	$-$ $-$	$\vee\vee$ $\vee\wedge$	$-$ $-$
$E2$	$-$ $\odot\vee$ $\odot\wedge$	$-$ $-$ $-$	$-$ $-$ $-$	$\odot\odot$ $\odot\vee$ $\odot\wedge$	$-$ $-$ $-$	$-$ $-$ $-$	$-$ $-$ $-$	$-$ $-$ $-$	$-$ $\odot\vee$ $\odot\wedge$	$\odot\odot$ $-$ $\odot\odot$

Table D.1: Summary of confluence case

rules	substitution	overlap	critical pair	common term	extra
$S1_\wedge$ $S2_\wedge$	$B' = \mathbf{t}, B$ $B'' = \mathbf{f}, B$	$\wedge(\mathbf{f}, \mathbf{t}, B)$	$\wedge(\mathbf{f})$ $\wedge(\mathbf{f}, B)$	$\wedge(\mathbf{f})$	
$S2_\vee$ $S3_\vee$	$B = [\![]\!]$ $\psi = \mathbf{f}$	$\vee(\mathbf{f})$	$\mathbf{f}$ $\mathbf{f}$	$\mathbf{f}$	
$P1_\wedge$ $P1_\wedge$	$B' = \alpha'', B$ $B'' = \alpha', B$	$\wedge(\alpha', \alpha'', B)$ $\{\alpha'' = \sim\alpha'\}$	$\wedge(\alpha', \mathbf{f}, B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\sim\alpha', \mathbf{f}, B[\mathbf{f} \stackrel{*\sim}{\leftarrow} \alpha'])$	$\wedge(\mathbf{f})$	$S1_\wedge$
$P1_\wedge$ $P1_\wedge$	$B' = \alpha'', B$ $B'' = \alpha', B$	$\wedge(\alpha', \alpha'', B)$ $\{\text{distinct } \alpha', \alpha''\}$	$\wedge(\alpha', \alpha'', B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \alpha'', B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	$\wedge(\alpha', \alpha'', B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'][\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha''])$	
$P1_\wedge$ $P1_\wedge$	$B' = \wedge(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \wedge(\alpha'', B''), B)$ $\{\alpha'' = \alpha'\}$	$\wedge(\alpha', \wedge(\mathbf{t}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \wedge(\mathbf{t}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	$\wedge(\alpha', \wedge(\mathbf{t}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	
$P1_\wedge$ $P1_\wedge$	$B' = \wedge(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \wedge(\alpha'', B''), B)$ $\{\alpha'' = \sim\alpha'\}$	$\wedge(\alpha', \wedge(\mathbf{f}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \wedge(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'']), B)$	$\wedge(\alpha', \wedge(\mathbf{f}), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	$S1_\wedge$
$P1_\wedge$ $P1_\wedge$	$B' = \wedge(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \wedge(\alpha'', B''), B)$ $\{\text{distinct } \alpha'', \alpha'\}$	$\wedge(\alpha', \wedge(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \wedge(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'']), B)$	$\wedge(\alpha', \wedge(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'][\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	
$P1_\wedge$ $P1_\vee$	$B' = \vee(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \vee(\alpha'', B''), B)$ $\{\alpha'' = \alpha'\}$	$\wedge(\alpha', \vee(\mathbf{t}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \vee(\alpha', B''[\mathbf{f} \stackrel{*\sim}{\leftarrow} \alpha'], B))$	$\wedge(\alpha, \vee(\mathbf{t}), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	$S1_\vee$
$P1_\wedge$ $P1_\vee$	$B' = \vee(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \vee(\alpha'', B''), B)$ $\{\alpha'' = \sim\alpha'\}$	$\wedge(\alpha', \vee(\mathbf{f}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$ $\wedge(\alpha', \vee(\alpha'', B''[\mathbf{f} \stackrel{*\sim}{\leftarrow} \alpha'']), B)$	$\wedge(\alpha', \vee(\mathbf{f}, B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']), B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'])$	
$P1_\wedge$ $P1_\vee$	$B' = \vee(\alpha'', B''), B$ $\{\text{variable}\}$	$\wedge(\alpha', \vee(\alpha'', B''), B)$ $\{\text{distinct } \alpha'', \alpha'\}$	$\wedge(\alpha', \vee(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'], B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']))$ $\wedge(\alpha', \vee(\alpha'', B''[\mathbf{f} \stackrel{*\sim}{\leftarrow} \alpha'']), B)$	$\wedge(\alpha', \vee(\alpha'', B''[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha'][\mathbf{f} \stackrel{*\sim}{\leftarrow} \alpha''], B[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha']))$	

Table D.2: Simplification and propagation rules

rules	substitution	overlap	critical pair	common term	extra
$P1_\wedge$ $S1_\wedge$	$B' = \mathbf{f}, B$ $B'' = \alpha, B$	$\wedge(\alpha, \mathbf{f}, B)$	$\wedge(\alpha, \mathbf{f}, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(f)$	$\wedge(\mathbf{f})$	
$P1_\wedge$ $S1_\wedge$	$B' = \wedge(\mathbf{f}, B''), B$ {variable}	$\wedge(\alpha, \wedge(\mathbf{f}, B''), B)$ $\{\alpha \text{ not in } B\}$	$\wedge(\alpha, \wedge(\mathbf{f}, B''[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B))$ $\wedge(\alpha, \wedge(\mathbf{f}), B)$	$\wedge(\alpha, \wedge(\mathbf{f}), B)$	
$P1_\wedge$ $S1_\vee$	$B' = \vee(\mathbf{t}, B''), B$ {variable}	$\wedge(\alpha, \vee(\mathbf{t}, B''), B)$ $\{\alpha \text{ not in } B\}$	$\wedge(\alpha, \vee(\mathbf{t}, B''[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B)$ $\wedge(\alpha, \vee(\mathbf{t}), B)$	$\wedge(\alpha, \vee(\mathbf{t}), B)$	
$P1_\wedge$ $S2_\wedge$	$B' = \mathbf{t}, B$ $B'' = \alpha, \mathbf{t}$	$\wedge(\alpha, \mathbf{t}, B)$	$\wedge(\alpha, \mathbf{t}, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(\alpha, B)$	$\wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	
$L1_\wedge$ $L1_\wedge$	$B_1' = \alpha'', B_1$ $B_1'' = \alpha', B_1$	$\wedge(\wedge(\alpha', \alpha'', B_1), B_2)$	$\wedge(\alpha', \wedge(\alpha'', B_1), B_2)$ $\wedge(\alpha'', \wedge(\alpha', B_1), B_2)$	$\wedge(\alpha', \alpha'', \wedge(B_1), B_2)$	
$L1_\wedge$ $L1_\wedge$	$B_2' = \wedge(\alpha'', B_1''), B_2$ $B_2'' = \wedge(\alpha', B_1'), B_2$	$\wedge(\wedge(\alpha', B_1'), \wedge(\alpha'', B_1''), B_2)$	$\wedge(\alpha', \wedge(B_1'), \wedge(\alpha'', B_1''), B_2)$ $\wedge(\wedge(\alpha', B_1'), \alpha'', \wedge(B_1''), B_2)$	$\wedge(\alpha', \alpha'', \wedge(B_1'), \wedge(B_1''), B_2)$	
$L1_\wedge$ $L2_\wedge$	$B_2 = \alpha_1, \ldots, \alpha_n$ $B = \wedge(\alpha, B_1)$	$\wedge(\wedge(\alpha, B_1), \alpha_1, \ldots, \alpha_n)$ symmetric	$\wedge(\alpha, \alpha_1, \ldots, \alpha_n, \wedge(B_1))$ $\wedge(\alpha, \alpha_1, \ldots, \alpha_n, B_1)$	$\wedge(\alpha, \alpha_1, \ldots, \alpha_n, B_1)$	
$L1_\wedge$ $L2_\wedge$	$B_1 = \alpha_2, \ldots, \alpha_n$ $\{\alpha = \alpha_1\}$	$\wedge(\wedge(\alpha_1, \ldots, \alpha_n, \wedge(B)), B_2)$	$\wedge(\alpha_1, \wedge(\alpha_2, \ldots, \alpha_n, \wedge(B)), B_2)$ $\wedge(\wedge(\alpha_1, \ldots, \alpha_n, B), B_2)$	$\wedge(\alpha_1, \wedge(\alpha_2, \ldots, \alpha_n, B), B_2)$	

Table D.3: Simplification, propagation, and lifting rules

rules	substitution	overlap	critical pair	common term	extra
$L1_\wedge$ $S1_\wedge$	$B_1 = \mathbf{f}, B, B'$ $B = \alpha, B_1, B'$	$\wedge(\wedge(\alpha, \mathbf{f}, B'), B_2)$	$\wedge(\alpha, \wedge(\mathbf{f}, B'), B_2)$ $\wedge(\wedge(\mathbf{f}), B_2)$	$\wedge(\mathbf{f})$	$S3_\wedge$
$L1_\wedge$ $S1_\wedge$	$B = \wedge(\alpha, B_1), B'$ $B_2 = \mathbf{f}, B'$	$\wedge(\wedge(\alpha, B_1), \mathbf{f}, B')$	$\wedge(\alpha, \wedge(B_1), \mathbf{f}, B')$ $\wedge(\mathbf{f})$	$\wedge(\mathbf{f})$	
$L1_\vee$ $S2_\vee$	$B_1 = \mathbf{f}, B, B'$ $B = \alpha, B_1, B'$	$\vee(\vee(\alpha, \mathbf{f}, B'), B_2)$	$\vee(\alpha, \vee(\mathbf{f}, B'), B_2)$ $\vee(\vee(\alpha, B'), B_2)$	$\vee(\alpha, \vee(B'), B_2)$	
$L1_\vee$ $S2_\vee$	$B = \vee(\alpha, B_1), B'$ $B_2 = \mathbf{f}, B'$	$\vee(\vee(\alpha, B_1), \mathbf{f}, B')$	$\vee(\alpha, \vee(B_1), \mathbf{f}, B')$ $\vee(\vee(\alpha, B_1), B')$	$\vee(\alpha, \vee(B_1), B')$	
$L1_\wedge$ $S3_\wedge$	$\psi = \wedge(\alpha, B_1)$ $B_2 = [\![\,]\!]$	$\wedge(\wedge(\alpha, B_1))$	$\wedge(\alpha, \wedge(B_1))$ $\wedge(\alpha, B_1)$	$\wedge(\alpha, B_1)$	$L2_\wedge$
$L1_\wedge$ $S3_\wedge$	$B_1 = [\![\,]\!]$ $\psi = \alpha$	$\wedge(\wedge(\alpha), B_2)$	$\wedge(\alpha, \mathbf{t}, B_2)$ $\wedge(\alpha, B_2)$	$\wedge(\alpha, B_2)$	$S2_\wedge$
$L2_\wedge$ $S1_\wedge$	$B' = \mathbf{f}, B''$	$\wedge(\alpha_1, \ldots, \alpha_n, \wedge(\mathbf{f}, B''))$	$\wedge(\alpha_1, \ldots, \alpha_n, \mathbf{f}, B'')$ $\wedge(\alpha_1, \ldots, \alpha_n, \wedge(\mathbf{f}))$	$\wedge(\mathbf{f})$	$S3_\wedge$
$L2_\vee$ $S2_\vee$	$B' = \mathbf{f}, B''$	$\vee(\alpha_1, \ldots, \alpha_n, \vee(\mathbf{f}, B''))$	$\vee(\alpha_1, \ldots, \alpha_n, \mathbf{f}, B'')$ $\vee(\alpha_1, \ldots, \alpha_n, \vee(B''))$	$\vee(\alpha_1, \ldots, \alpha_n, B'')$	
$L2_\vee$ $S2_\vee$	$B' = [\![\,]\!]$ $B'' = \alpha_1, \ldots, \alpha_n$	$\vee(\alpha_1, \ldots, \alpha_n, \mathbf{f})$	$\vee(\alpha_1, \ldots, \alpha_n)$ $\vee(\alpha_1, \ldots, \alpha_n)$	$\vee(\alpha_1, \ldots, \alpha_n)$	
$L2_\wedge$ $S3_\wedge$	$\psi = \wedge(B)$ $n = 0$	$\wedge(\wedge(B))$	$\wedge(B)$ $\wedge(B)$	$\wedge(B)$	
$L2_\wedge$ $S3_\wedge$	$\psi = B$	$\wedge(\alpha_1, \ldots, \alpha_n, \wedge(\psi))$	$\wedge(\alpha_1, \ldots, \alpha_n, \psi)$ $\wedge(\alpha_1, \ldots, \alpha_n, \psi)$	$\wedge(\alpha_1, \ldots, \alpha_n, \psi)$	

Table D.4: Lifting and simplification rules

rules	substitution	overlap	critical pair	common term	extra
$L1_\wedge$ $P1_\wedge$	$B_1 = B$	$\wedge(\wedge(\alpha, B), B_2)$	$\wedge(\alpha, \wedge(B), B_2)$ $\wedge(\wedge(\alpha, B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2)$	$\wedge(\alpha, \wedge(B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	
$L1_\wedge$ $P1_\wedge$	$B = \wedge(\alpha', B_1), B_2'$ $B_2 = B_2', \alpha''$	$\wedge(\alpha'', \wedge(\alpha', B_1), B_2')$ $\{\alpha' = \alpha''\}$	$\wedge(\alpha'', \alpha', \wedge(B_1), B_2')$ $\wedge(\alpha', \wedge(\mathbf{t}, B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha']), B_2'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'])$	$\wedge(\alpha', \wedge(B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha']), B_2'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'])$	$S2_\wedge$
$L1_\wedge$ $P1_\wedge$	$B = \wedge(\alpha', B_1), B_2'$ $B_2 = B_2', \alpha''$	$\wedge(\alpha'', \wedge(\alpha', B_1), B_2')$ $\{\alpha' =\sim \alpha''\}$	$\wedge(\alpha'', \alpha', \wedge(B_1), B_2')$ $\wedge(\alpha'', \wedge(\mathbf{f}, B_1[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha']), B_2'[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha'])$	$\mathbf{f}$	$S1_\wedge$ $S3_\wedge$
$L1_\wedge$ $P1_\wedge$	$B = \wedge(\alpha', B_1), B_2'$ $B_2 = B_2', \alpha''$	$\wedge(\alpha'', \wedge(\alpha', B_1), B_2')$ $\{\text{distinct } \alpha', \alpha''\}$	$\wedge(\alpha'', \alpha', \wedge(B_1), B_2')$ $\wedge(\alpha'', \wedge(\alpha', B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']), B_2'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''])$	$\wedge(\alpha'', \alpha', \wedge(B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']), B_2'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''])$	
$P1_\wedge$ $L1_\wedge$	$B = \wedge(\wedge(\alpha'', B_1), B_2)$ $\{\text{variable}\}$	$\wedge(\alpha, \wedge(\wedge(\alpha'', B_1), B_2))$ $\{\alpha = \alpha' = \alpha''\}$	$\wedge(\alpha, \wedge(\wedge(\mathbf{t}, B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$ $\wedge(\alpha, \wedge(\alpha'', \wedge(B_1), B_2))$	$\wedge(\alpha, \wedge(\wedge(B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	$S2_\wedge$
$P1_\wedge$ $L1_\wedge$	$B = \wedge(\wedge(\alpha'', B_1), B_2)$ $\{\text{variable}\}$	$\wedge(\alpha, \wedge(\wedge(\alpha'', B_1), B_2))$ $\{\alpha = \alpha' =\sim \alpha''\}$	$\wedge(\alpha, \wedge(\wedge(\mathbf{f}, B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$ $\wedge(\alpha, \wedge(\alpha'', \wedge(B_1), B_2))$	$\wedge(\alpha, \mathbf{f})$	$S1_\wedge$ $S3_\wedge$
$P1_\wedge$ $L1_\vee$	$B = \vee(\vee(\alpha'', B_1), B_2)$ $\{\text{variable}\}$	$\wedge(\alpha, \vee(\vee(\alpha'', B_1), B_2))$ $\{\alpha = \alpha' = \alpha''\}$	$\wedge(\alpha, \vee(\vee(\mathbf{t}, B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$ $\wedge(\alpha, \vee(\alpha'', \vee(B_1), B_2))$	$\wedge(\alpha, \vee(\mathbf{t}))$	$S1_\vee$ $S3_\vee$
$P1_\wedge$ $L1_\vee$	$B = \vee(\vee(\alpha'', B_1), B_2)$ $\{\text{variable}\}$	$\wedge(\alpha, \vee(\vee(\alpha'', B_1), B_2))$ $\{\alpha = \alpha' =\sim \alpha''\}$	$\wedge(\alpha, \vee(\vee(\mathbf{f}, B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$ $\wedge(\alpha, \vee(\alpha'', \vee(B_1), B_2))$	$\wedge(\alpha, \vee(\vee(B_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B_2[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	$S2_\vee$

Table D.5: Lifting and propagation rules

rules	substitution	overlap	critical pair	common term	extra
$L2_\wedge$ $P1_\wedge$	$\alpha = \alpha_1$ $B'' = \alpha_2, \ldots, \alpha_n, \wedge(B')$	$\wedge(\overline{\alpha_i}, \wedge(B))$ $\alpha_2 =\sim \alpha$	$\wedge(\overline{\alpha_i}, B)$ $\wedge(\alpha, \alpha_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \ldots, \alpha_n[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \wedge(B'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	$\wedge(\mathbf{f})$	$S1_\wedge$
$L2_\wedge$ $P1_\wedge$	$\alpha = \alpha_1$ $B'' = \alpha_2, \ldots, \alpha_n, \wedge(B')$	$\wedge(\overline{\alpha_i}, \wedge(B))$ {otherwise}	$\wedge(\overline{\alpha_i}, B)$ $\wedge(\alpha, \alpha_1[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \ldots, \alpha_n[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \wedge(B'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	$\wedge(\alpha, \{\alpha_i \mid \alpha_1 \neq \alpha\}, \wedge(B'[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	$S2_\wedge$
$P1_\wedge$ $L2_\vee$	$B' = \vee(\overline{\alpha_i}, \vee(B'')), B$ {variable}	$\wedge(\alpha, \vee(\overline{\alpha_i}, \vee(B'')), B)$ $\alpha = \alpha_1$	$\wedge(\alpha, \vee(\overline{\alpha_i}, \vee(B''))[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(\alpha, \vee(\overline{\alpha_i}, B''), B)$	$\wedge(\alpha, \vee(\mathbf{t}), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	$S1_\vee$
$P1_\wedge$ $L2_\vee$	$B' = \vee(\overline{\alpha_i}, \vee(B'')), B$ {variable}	$\wedge(\alpha, \vee(\overline{\alpha_i}, \vee(B'')), B)$ {otherwise}	$\wedge(\alpha, \vee(\overline{\alpha_i}, \vee(B''))[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(\alpha, \vee(\overline{\alpha_i}, B''), B)$	$\wedge(\alpha, \vee(\{\alpha_i \mid \alpha_1 \neq \alpha\}, B''[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	$S2_\vee$
$P1_\wedge$ $L2_\wedge$	$B' = \wedge(\overline{\alpha_i}, \wedge(B'')), B$ {variable}	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \wedge(B'')), B)$ $\alpha =\sim \alpha_1$	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \wedge(B'')[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(\alpha, \wedge(\overline{\alpha_i}, B''), B)$	$\wedge(\alpha, \wedge(\mathbf{f}), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	$S1_\wedge$
$P1_\wedge$ $L2_\wedge$	$B' = \wedge(\overline{\alpha_i}, \wedge(B'')), B$ {variable}	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \wedge(B'')), B)$ {otherwise}	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \wedge(B'')[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$ $\wedge(\alpha, \wedge(\overline{\alpha_i}, B''), B)$	$\wedge(\alpha, \wedge(\{\alpha_i \mid \alpha_1 \neq \alpha\}, B''[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), B[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$	$S2_\wedge$

Table D.6: Propagation and lifting rules

rules	substitution	overlap
extra	critical pair	common term
$F1_\wedge$	{substitution}	$\wedge(\overline{\alpha_i}, \vee(\alpha', \alpha'', B_0), \ldots, \vee(\alpha', \alpha'', B_m))$ {overlap}
$S3_\wedge$	$\wedge(\overline{\alpha_i}, \vee(\alpha', \wedge(\vee(\alpha'', B_0), \ldots, \vee(\alpha'', B_m))))$ {CP-1}	$\wedge(\overline{\alpha_i}, \vee(\alpha', \alpha'', \wedge(\vee(B_0), \ldots, \vee(B_m))))$ { common}
....	....	....
$F1_\wedge$		
$L1_\vee$	$\wedge(\overline{\alpha_i}, \vee(\alpha'', \wedge(\vee(\alpha', B_0), \ldots, \vee(\alpha', B_m))))$ {CP-2}	
$F1_\wedge$		$\wedge(\overline{\alpha_i}', \vee(\overline{\alpha_i}'', \alpha', \wedge(\alpha'', B_0''), \ldots, \wedge(\alpha'', B''_{m''})), \vee(\alpha', B_1'), \ldots, \vee(\alpha', B'_{m'}))$
	$\wedge(\overline{\alpha_i}', \vee(\alpha', \wedge(\vee(\overline{\alpha_i}'', \wedge(\alpha'', B_0''), \ldots, \wedge(\alpha'', B''_{m''})), \vee(B_1'), \ldots, \vee(B'_{m'}))))$	$\wedge(\overline{\alpha_i}', \vee(\alpha', \wedge(\vee(\overline{\alpha_i}'', \wedge(\alpha'', \vee(\wedge(B_0''), \ldots, \wedge(B''_{m''}))))), \vee(B_1'), \ldots, \vee(B'_{m'}))))$
....	....	....
$F1_\vee$		
	$\wedge(\overline{\alpha_i}', \vee(\overline{\alpha_i}'', \alpha', \wedge(\alpha'', \vee(\wedge(B_0''), \ldots, \wedge(B''_{m''})))), \vee(\alpha', B_1'), \ldots, \vee(\alpha', B'_{m'}))$	
$F1_\vee$	$\alpha = a \doteq a$	$\vee(\overline{\alpha_i}, \wedge(a \doteq a, B_0), \ldots, \wedge(a \doteq a, B_m))$
$S2_\wedge, S3_\wedge$	$\vee(\overline{\alpha_i}, \wedge(a \doteq a, \vee(\wedge(B_0), \ldots, \wedge(B_m))))$	$\vee(\overline{\alpha_i}, \wedge(B_0), \ldots, \wedge(B_m))$
....	....	....
$E1_\wedge$		
$L2_\vee$	$\vee(\overline{\alpha_i}, \wedge(\mathbf{t}, B_0), \ldots, \wedge(\mathbf{t}, B_m))$	
$F1_\vee$	$\alpha = a \dot{\neq} a$	$\vee(\overline{\alpha_i}, \wedge(a \dot{\neq} a, B_0), \ldots, \wedge(a \dot{\neq} a, B_m))$
$S1_\wedge, S2_\vee$	$\vee(\overline{\alpha_i}, \wedge(a \dot{\neq} a, \vee(\wedge(B_0), \ldots, \wedge(B_m))))$	$\vee(\overline{\alpha_i}, \mathbf{f})$
....	....	....
$E1_\vee$		
$S3_\wedge$	$\vee(\overline{\alpha_i}, \wedge(\mathbf{f}, B_0), \ldots, \wedge(\mathbf{f}, B_m))$	

Table D.7: Factoring and equality rules : $m, m', m'' \geq 1$

rules extra	substitution critical pair	overlap common term
$F1_\vee$		$\vee(\overline{\alpha_i}, \wedge(\alpha, \mathbf{f}, B_0), \wedge(\alpha, B_1))$
$S2_\vee, S3_\vee$	$\vee(\overline{\alpha_i}, \wedge(\alpha, \vee(\wedge(\mathbf{f}, B_0), \wedge(B_1))))$	$\vee(\overline{\alpha_i}, \wedge(\alpha, B_1))$
....	....	....
$S1_\wedge$		
$S3_\wedge, L2_\wedge$	$\vee(\overline{\alpha_i}, \wedge(\mathbf{f}), \wedge(\alpha, B_1))$	
$F1_\vee$		$\vee(\overline{\alpha_i}, \wedge(\alpha, \mathbf{f}, B_0), \wedge(\alpha, B_1), \ldots, \wedge(\alpha, B_m))$
$S2_\vee$	$\vee(\overline{\alpha_i}, \wedge(\alpha, \vee(\wedge(\mathbf{f}, B_0), \wedge(B_1), \ldots, \wedge(B_m))))$	$\vee(\overline{\alpha_i}, \wedge(\alpha, \vee(\wedge(B_1), \ldots, \wedge(B_m))))$
....	....	....
$S1_\wedge$		$\{m > 1\}$
$S3_\wedge$	$\vee(\overline{\alpha_i}, \wedge(\mathbf{f}), \wedge(\alpha, B_1), \ldots, \wedge(\alpha, B_m))$	
$F1_\wedge$		$\wedge(\overline{\alpha_i}, \vee(\alpha, \mathbf{f}, B_0), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$
	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(\mathbf{f}, B_0), \vee(B_1), \ldots, \vee(B_m))))$	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(B_0), \vee(B_1), \ldots, \vee(B_m))))$
....	....	....
$S2_\vee$		
	$\wedge(\overline{\alpha_i}, \vee(\alpha, B_0), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$	
$F1_\wedge$		$\wedge(\overline{\alpha_i}, \vee(\alpha), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$
$S1_\vee, S1_\wedge$	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(), \vee(B_1), \ldots, \vee(B_m))))$	$\wedge(\overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \alpha)$
....	....	....
$S3_\vee$		
$S2_\vee, P1_\wedge$	$\wedge(\overline{\alpha_i}, \alpha, \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$	

Table D.8: Factoring and simplification rules: $m, m', m'' \geq 1$

rules extra	substitution critical pair	overlap common term
$F1_\wedge$	$\alpha = \alpha' = \alpha''$	$\wedge(\alpha, \overline{\alpha_i}, \vee(\alpha, B_0), \ldots, \vee(\alpha, B_m))$
$S1_\vee, S2_\wedge$	$\wedge(\alpha, \overline{\alpha_i}, \vee(\alpha, \wedge(\vee(B_0), \ldots, \vee(B_m))))$	$\wedge(\alpha, \overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])$
....	....	....
$P1_\wedge$	$B = \overline{\alpha_i}, \vee(\alpha, B_0), \ldots, \vee(\alpha, B_m)$	
$S3_\vee$	$\wedge(\alpha, \overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \vee(\mathbf{t}, B_0[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), \ldots, \vee(\mathbf{t}, B_m[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$	
$F1_\wedge$	$\sim \alpha' = \alpha''$	$\wedge(\alpha'', \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m))$
$S2_\vee, S3_\vee$	$\wedge(\alpha'', \overline{\alpha_i}, \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m))))$	$\wedge(\alpha'', \overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''], \vee(B_0[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']), \ldots, \vee(B_m[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']))$
....	....	....
$P1_\wedge$	$B = \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)$	
$L2_\wedge$	$\wedge(\alpha'', \overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''], \vee(\mathbf{f}, B_0[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']), \ldots, \vee(\mathbf{f}, B_m[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']))$	
$F1_\wedge$		$\wedge(\alpha'', \sim \alpha'', \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m))$
$S1_\wedge$	$\wedge(\alpha'', \sim \alpha'', \overline{\alpha_i}, \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m))))$	$\wedge(\mathbf{f})$
....	....	....
$P1_\wedge$		
	$\wedge(\alpha'', \mathbf{f}, \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m))$	
$F1_\wedge$		$\wedge(\alpha'', \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m))$
$S2_\wedge$	$\wedge(\alpha'', \overline{\alpha_i}, \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m))))$	$\wedge(\alpha'', \{\alpha_i \mid \alpha_i \neq \alpha''\}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''], \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m))[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha'']))$
....	....	....
$P1_\wedge$	$\{$distinct $\alpha', \alpha''\}$	$\{\sim \alpha''$ not in $\overline{\alpha_i}\}$
	$\wedge(\alpha'', \overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''], \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha''])$	

Table D.9: Factoring and propagation rules I: $m \geq 1$

rules extra	substitution critical pair	overlap common term
$F1_\wedge$		$\wedge(\overline{\alpha_i}, \vee(\alpha, B_0), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$
	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(B_0), \vee(B_1), \ldots, \vee(B_m))))$	$\wedge(\vee(\alpha, \wedge(\vee(B_0[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]), \vee(B_1[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]), \ldots, \vee(B_m[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]))))$
....	....	....
$P1_\vee$		
	$\wedge(\overline{\alpha_i}, \vee(\alpha, B_0[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$	
$F1_\wedge$		$\wedge(\overline{\alpha_i}, \vee(\alpha, \sim\alpha, B_0), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$
$S1_\vee, S2_\wedge$	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(\sim\alpha, B_0), \vee(B_1), \ldots, \vee(B_m))))$	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(B_0[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]), \ldots, \vee(B_m[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]))))$
....	....	....
$P1_\vee$	$\{m > 1\}$	
$S3_\vee$	$\wedge(\overline{\alpha_i}, \vee(\mathbf{t}, \sim\alpha, B_0), \vee(\alpha, B_1), \ldots, \vee(\alpha, B_m))$	
$F1_\wedge$		$\wedge(\overline{\alpha_i}, \vee(\alpha, \sim\alpha, B_0), \vee(\alpha, B_1))$
$S1_\vee, S2_\wedge, S3_\wedge$	$\wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(\sim\alpha, B_0), \vee(B_1))))$	$\wedge(\overline{\alpha_i}, \vee(\alpha, B_0[\mathbf{f} \overset{*\sim}{\leftarrow} \alpha]))$
....	....	....
$P1_\vee$		
$S3_\vee, L2_\vee$	$\wedge(\overline{\alpha_i}, \vee(\mathbf{t}, \sim\alpha, B_0), \vee(\alpha, B_1))$	
$P1_\wedge$		$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\alpha, B_0), \ldots, \vee(\alpha, B_m)))$
$S1_\vee, S2_\wedge$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha], \vee(\mathbf{t}, B_0[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]), \ldots, \vee(\mathbf{t}, B_m[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha])))$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \overset{*\sim}{\leftarrow} \alpha]))$
....	....	....
$F1_\wedge$		
$S3_\vee$	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\alpha, \wedge(\vee(B_0), \ldots, \vee(B_m)))))$	

Table D.10: Factoring and propagation rules II: $m \geq 1$

rules extra	substitution critical pair	overlap common term
$P1_\wedge$		$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\sim a, B_0), \ldots, \vee(\sim a, B_m)))$
$S2_\vee$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha], \vee(\mathbf{f}, B_0[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha]), \ldots, \vee(\mathbf{f}, B_m[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha])))$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha], \vee(B_0[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha]), \ldots, \vee(B_m[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha])))$
....	....	....
$F1_\wedge$		
$L2_\wedge$	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\sim \alpha, \wedge(\vee(B_0), \ldots, \vee(B_m)))))$	
$P1_\wedge$		$\wedge(\alpha, \wedge(\sim \alpha, \overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)))$
$S1_\wedge$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha], \mathbf{f}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha]))$	$\wedge(\alpha, \wedge(\mathbf{f}))$
....	....	....
$F1_\wedge$		
	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \sim \alpha, \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m)))))$	
$P1_\wedge$		$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)))$
$S2_\wedge$	$\wedge(\alpha, \wedge(\overline{\alpha_i}[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha], \vee(\alpha', B_0), \ldots, \vee(\alpha', B_m)[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha]))$	$\wedge(\alpha, \{\alpha_i \mid \alpha_i \neq \alpha\}[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha], \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m)))[\mathbf{t} \stackrel{*\sim}{\leftarrow} \alpha])$
....	....	....
$F1_\wedge$		
	$\wedge(\alpha, \wedge(\overline{\alpha_i}, \vee(\alpha', \wedge(\vee(B_0), \ldots, \vee(B_m)))))$	

Table D.11: Propagation and factoring rules

rules extra	substitution critical pair	overlap common term
$F1_\wedge$ .... $L1_\vee$	$B'_0 = \vee(\alpha'', B''_1), B''_2$ $\wedge(\overline{\alpha_i}, \vee(\alpha', \wedge(\vee(\vee(\alpha'', B''_1), B''_2), \vee(B'_1), \ldots, \vee(B'_m))))$ .... $\wedge(\overline{\alpha_i}, \vee(\alpha', \alpha'', \vee(B''_1), B''_2), \vee(\alpha', B'_1), \ldots, \vee(\alpha', B'_m))$	$\wedge(\overline{\alpha_i}, \vee(\alpha', \vee(\alpha'', B''_1), B''_2), \vee(\alpha', B'_1), \ldots, \vee(\alpha', B'_m))$ $\wedge(\overline{\alpha_i}, \vee(\alpha', \wedge(\vee(\alpha'', \vee(B''_1), B''_2), \vee(B'_1), \ldots, \vee(B'_m))))$ ....
$L1_\wedge$ .... $F1_\wedge$	$B'_1 = \overline{\alpha_i}, \vee(\alpha'', B''_0), \ldots, \vee(\alpha'', B''_m)$ $\wedge(\wedge(\alpha', \overline{\alpha_i}, \vee(\alpha'', \wedge(\vee(B''_0), \ldots, \vee(B''_m)))), B'_2)$ .... $\wedge(\wedge(\alpha', \overline{\alpha_i}, \vee(\alpha'', \wedge(\vee(B''_0), \ldots, \vee(B''_m)))), B'_2)$	$\wedge(\wedge(\alpha', \overline{\alpha_i}, \vee(\alpha'', B''_0), \ldots, \vee(\alpha'', B''_m)), B'_2)$ $\wedge(\alpha', \wedge(\overline{\alpha_i}, \vee(\alpha'', \wedge(\vee(B''_0), \ldots, \vee(B''_m)))), B'_2)$ ....
$F1_\wedge$ .... $L2_\vee$	 $\wedge(\overline{\alpha_i}', \vee(\alpha', \wedge(\vee(\overline{\alpha_i}'', \vee(B)), \vee(B_1), \ldots, \vee(B_m))))$ .... $\wedge(\overline{\alpha_i}', \vee(\alpha', \overline{\alpha_i}'', B), \vee(\alpha', B_1), \ldots, \vee(\alpha', B_m))$	$\wedge(\overline{\alpha_i}', \vee(\alpha', \overline{\alpha_i}'', \vee(B)), \vee(\alpha', B_1), \ldots, \vee(\alpha', B_m))$ $\wedge(\overline{\alpha_i}', \vee(\alpha', \wedge(\vee(\overline{\alpha_i}'', B), \vee(B_1), \ldots, \vee(B_m))))$ ....

Table D.12: Factoring and lifting rules

rules	overlap	critical pair	common term	extra
$E2_{\odot}$ $S1_{\wedge}$	$\odot(a \doteq b, \wedge(\mathbf{f}, B))$	$\odot(a \doteq b, \wedge(\mathbf{f}, B[b \overset{*}{\leftarrow} a]))$ $\odot(a \doteq b, \wedge(\mathbf{f}))$	$\odot(a \doteq b, \wedge(\mathbf{f}))$	
$E2_{\odot}$ $S1_{\vee}$	$\odot(a \doteq b, \vee(\mathbf{t}, B))$	$\odot(a \doteq b, \vee(\mathbf{t}, B[b \overset{*}{\leftarrow} a]))$ $\odot(a \doteq b, \vee(\mathbf{t}))$	$\odot(a \doteq b, \vee(\mathbf{t}))$	
$E2_{\odot}$ $P1_{\odot}$	$\odot(a \doteq b, B)$	$\odot(a \doteq b, B[b \overset{*}{\leftarrow} a])$ $\odot(a \doteq b, B[\mathbf{t} \overset{*\sim}{\leftarrow} a \doteq b])$	$\odot(a \doteq b, B[b \overset{*}{\leftarrow} a][\mathbf{t} \overset{*\sim}{\leftarrow} b \doteq b])$	$E1_{\wedge}$
$E2_{\odot}$ $P1_{\wedge}$	$\odot(a \doteq b, P(a,b), B)$	$\odot(a \doteq b, P(b,b), B[b \overset{*}{\leftarrow} a])$ $\odot(a \doteq b, P(a,b), B[\mathbf{t} \overset{*\sim}{\leftarrow} P(a,b)])$	$\odot(a \doteq b, P(b,b), B[b \overset{*}{\leftarrow} a][\mathbf{t} \overset{*\sim}{\leftarrow} P(b,b)])$	
$E2_{\odot}$ $P1_{\vee}$	$\odot(a \doteq b, \vee(P(a,b), B_1), B_2)$	$\odot(a \doteq b, \vee(P(b,b), B_1[b \overset{*}{\leftarrow} a]), B_2[b \overset{*}{\leftarrow} a])$ $\odot(a \doteq b, \vee(P(a,b), B_1[\mathbf{f} \overset{*\sim}{\leftarrow} P(a,b)]), B_2)$	$\odot(a \doteq b, \vee(P(b,b), B_1[b \overset{*}{\leftarrow} a][\mathbf{f} \overset{*\sim}{\leftarrow} P(b,b)])),$ $B_2[b \overset{*}{\leftarrow} a]$	
$P1_{\vee}$ $E1_{\wedge}$	$\vee(a \doteq a, B)$ $a \doteq a \in B$	$\vee(\mathbf{t}, B)$ $\vee(a \doteq a, B[\mathbf{f} \overset{*\sim}{\leftarrow} a \doteq a])$	$\vee(\mathbf{t})$	$S1_{\vee}$
$P1_{\wedge}$ $E1_{\wedge}$	$\wedge(a \doteq a, B)$ $a \doteq a \in B$	$\wedge(\mathbf{t}, B)$ $\wedge(a \doteq a, B[\mathbf{t} \overset{*\sim}{\leftarrow} a \doteq a])$	$\wedge(\mathbf{t}, B[\mathbf{t} \overset{*\sim}{\leftarrow} a \doteq a])$	

Table D.13: Equality rules I

rules	overlap	critical pair	common term	extra
$L2_\vee$ $E1_\wedge$	$\vee(a \doteq a, \alpha_2, \ldots, \alpha_n, \vee(B))$	$\vee(a \doteq a, \alpha_2, \ldots, \alpha_n, B)$ $\vee(\mathbf{t}, \alpha_2, \ldots, \alpha_n, \vee(B))$	$\vee(\mathbf{t})$	$S1_\vee$
$L2_\vee$ $E1_\vee$	$\vee(a \dot{\neq} a, \alpha_2, \ldots, \alpha_n, \vee(B))$	$\vee(a \dot{\neq} a, \alpha_2, \ldots, \alpha_n, B)$ $\vee(\mathbf{f}, \alpha_2, \ldots, \alpha_n, \vee(B))$	$\vee(\alpha_2, \ldots, \alpha_n, B)$	$S2_\vee$
$L1_\vee$ $E1_\wedge$	$\vee(\vee(a \doteq a, B_1), B_2)$	$\vee(\vee(\mathbf{t}, B_1), B_2)$ $\vee(a \doteq a, \vee(B_1), B_2)$	$\vee(\mathbf{t})$	$S1_\wedge$ $S2_\wedge$
$L1_\wedge$ $E1_\wedge$	$\wedge(\wedge(a \doteq a, B_1), B_2)$	$\wedge(\wedge(\mathbf{t}, B_1), B_2)$ $\wedge(a \doteq a, \wedge(B_1), B_2)$	$\wedge(\mathbf{t}, \wedge(B_1), B_2)$	$\vee(S2)$
$F1_\vee$ $E1_\wedge$	$\vee(\alpha_1, \ldots \alpha_n, \wedge(a \doteq a, B_0), \ldots, \wedge(a \doteq a, B_m))$	$\vee(\alpha_1, \ldots \alpha_n, \wedge(\mathbf{t}, B_0), \wedge(a \doteq a, B_1), \ldots, \wedge(a \doteq a, B_m))$ $\vee(\alpha_1, \ldots \alpha_n, \wedge(a \doteq a, \vee(\wedge(B_0), \ldots, \wedge(B_m))))$	$\vee(\alpha_1, \ldots \alpha_n, \wedge(B_0), \ldots, \wedge(B_m))$	$S2_\wedge$ $S2_\vee$
$F1_\vee$ $E1_\wedge$	$\vee(\alpha_1, \ldots \alpha_n, \wedge(a \dot{\neq} a, B_0), \ldots, \wedge(a \dot{\neq} a, B_m))$	$\vee(\alpha_1, \ldots \alpha_n, \wedge(\mathbf{f}, B_0), \wedge(a \dot{\neq} a, B_1), \ldots, \wedge(a \dot{\neq} a, B_m))$ $\vee(\alpha_1, \ldots \alpha_n, \wedge(a \dot{\neq} a, \vee(\wedge(B_0), \ldots, \wedge(B_m))))$	$\vee(\alpha_1, \ldots \alpha_n, \mathbf{f})$	$S1_\wedge, S2_\wedge$ $S2_\vee$
$E2_\odot$ $E1_\wedge$	$\odot(a \doteq b, a \doteq a)$	$\odot(a \doteq b, b \doteq b)$ $\odot(a \doteq b, \mathbf{t})$	$\odot(a \doteq b, \mathbf{t})$	
$E2_\odot$ $E1_\vee$	$\odot(a \doteq b, a \dot{\neq} a)$	$\odot(a \doteq b, b \dot{\neq} b)$ $\odot(a \doteq b, \mathbf{f})$	$\odot(a \doteq b, \mathbf{f})$	
$E2_\odot$ $E2_\odot$	$\odot(a \doteq b, a \doteq c, B)$	$\odot(a \doteq b, b \doteq c, B[b \overset{*}{\leftarrow} a])$ $\odot(c \doteq b, a \doteq c, B[c \overset{*}{\leftarrow} a])$	$\odot(a \doteq c, b \doteq c, B[b \overset{*}{\leftarrow} a][c \overset{*}{\leftarrow} b])$	

Table D.14: Equality rules II: $m > 0$ and $a \succ b \succ c$

Bibliography

[AB75] A.R. Anderson and N.D Belnap. *Entailment, the logic of relevance and neccessity.* Princeton University Press, 1975.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.

[Arn85] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability — a survey. *BIT*, 25:2–23, 1985.

[BB70] G. Birkhoff and T.C. Bartee. *Modern Applied Algebra.* McGraw-Hill, New York, 1970.

[BD88] M. Boddy and T. Dean. Solving time dependent planning problems. Technical report, Dept. of Computer Science, Brown University, 1988.

[BE89] A. Borgida and D.W. Etherington. Hierarchical knowledge bases and efficient disjunctive reasoning. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, pages 33–43. Morgan Kaufmann, 1989.

[Bel77] N. D. Belnap. A useful four-valued logic. In G. Epstein and J. M. Dunn, editors, *Modern Uses of Multiple-Valued Logics.* Reidel, 1977.

[BFL83] R.J. Brachman, R.E. Fikes, and H.J. Levesque. Krypton: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, 1983.

[Bun92] H.K. Buning. On generalized Horn formulas and k-resolution. *Information Processing Letters*, 1992. To be published.

[CK89] J. Crawford and B. Kuipers. Towards a theory of access-limited logic for knowledge representation. In *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 67–78, 1989.

[CK91] J.M. Crawford and B.J. Kuipers. Negation and proof by contradiction in access-limited logic. In *Proceedings Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 897–903, 1991.

[CKT91] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings Twelveth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–412, 1991.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, London, 1973.

[Coo71] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[Coo76] S. A. Cook. A short proof of the pigeon hole principle using extended resolution. *ACM SIGACT News*, 8:28–32, Oct.-Dec. 1976.

[Cra92] J. Crawford, editor. *Proceedings of the AAAI Workshop on Tractable Reasoning.* American Association for Artificial Intelligence, San Jose, California, 1992.

[Cra94] J. M. Crawford. Personal Communication, 1994.

[CS88] Chvatal and Szemeredi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759, oct. 1988.

[CS92a] M. Cadoli and M. Schaerf. Approximation in concept description languages. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 330–341, Cambridge, Massachusetts, 1992. Morgan Kaufmann Publishers.

[CS92b] M. Cadoli and M. Schaerf. Tractable reasoning via approximation. In *[Cra92]*, pages 12–15, 1992.

[Dav91] E. Davis. Lucid representations. Technical Report 565, New York University, Dept. of Computer Science, June 1991.

[de 89] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 290–296, 1989.

[de 90] J. de Kleer. Exploiting locality in a TMS. In *Proceedings Eight National Conference on Artificial Intelligence (AAAI-90)*, pages 264–271, 1990.

[DE92] M. Dalal and D. W. Etherington. Tractable approximate deduction using limited vocabularies. In *Proceedings Ninth Canadian Conference on Artificial Intelligence (AI '92)*, pages 206–212, Vancouver, Canada, 1992.

[Dec91] R. Dechter. Constraint satisfaction. In S.C. Shapiro, editor, *Encyclopedia of AI (2nd Edition)*. John Wiley and Sons, 1991.

[Der89] N. Dershowitz. Completion and its applications. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, chapter 2, pages 31–85. Academic Press, Inc., 1989.

[DG84] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[DH91] Y. Deville and P.V. Hentenryck. An efficient arc consistency algorithm for a class of CSP problems. In *Proceedings Twelveth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 325–330, 1991.

[DJ90] N. Dershowitz and J.-P. Jounnaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. The MIT Press, 1990.

[DP87] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[DP88] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.

[DP91] J. Doyle and R. Patil. Two theses of knowledge representation: language restrictions, taxanomic classification, and the utility of representation services. *Artificial Intelligence*, 48(3):261–297, 1991.

[Fit90] M. Fitting. *First-order logic and automated theorem proving.* Texts and monographs in computer science. Springer-Verlag, 1990.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Fre82] E.C. Freuder. A sufficient condition for backtract-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

[Fre85] E.C. Freuder. A sufficient condition for backtract-bounded search. *Journal of the ACM*, 32(4):755–761, october 1985.

[Fri87] A.M. Frisch. Inference without chaining. In *Proceedings Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 515–519, 1987.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, W. H., NY, 1979.

[GS88] G. Gallo and M.G. Scutellà. Polynomially solvable satisfiability problems. *Information Processing Letters*, 29:221–227, 1988.

[GTT92] T. Griffin, H. Trickey, and C. Tuckey. Update constraints for relational databases. unpublished draft, 1992.

[Hue80] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[IL82] T. Imielinski and W. Lipski. A systematic approach to relational database theory. In *SIGMOD*, 1982.

[Imi87] T. Imielinski. Domain abstraction and limited reasoning. In *Proceedings Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 997–1003, Milan, 1987.

[IMV94] T. Imielinski, R. v. d. Meyden, and K. V. Vadaparty. Complexity tailored design: A new design methodology for databases with incomplete information. To appear in Journal of Computer and System Sciences (A preliminary version appeared in PODS-89), 1994.

[KB70] D.E. Knuth and P. E. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[LB85] H.J. Levesque and R.J. Brachman. A fundamental tradeoff in knowledge representation and reasoning (revised version). In R.J. Brachman and H.J. Levesque, editors, *Readings in Knowledge Representation*, pages 41–70. Morgan Kaufmann, Los Altos, California, 1985.

[Lev84a] H.J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212, 1984.

[Lev84b] H.J. Levesque. A logic of implicit and explicit belief. *Proceedings National Conference on Artificial Intelligence (AAAI-84)*, pages 198–202, 1984.

[Lev86] H.J. Levesque. Making believers out of computers. *Artificial Intelligence*, 30:81–108, 1986.

[LJR92] J. Lobo, Minker J., and A. Rajasekar. *Foundations of disjunctive logic programming.* MIT Press, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2 edition, 1987.

[Mac77] A. K. Macworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mac87] A.K. Mackworth. Constraint satisfaction. In S.C. Shapiro, editor, *Encyclopedia of AI*, pages 205–211. John Wiley and Sons, 1987.

[Mar91] J. P. Martins. The truth, the whole truth, and nothing but the truth: An indexed bibliography to the literature of Truth Maintenance Systems. *The AI Magazine*, 11(5):7–25, January 1991.

[McA80] D. McAllester. An outlook on truth maintenance. Memo 551, MIT AI Lab, August 1980.

[McA90] D. McAllester. Truth maintenance. In *Proceedings Eight National Conference on Artificial Intelligence (AAAI-90)*, pages 1109–1116, 1990.

[McC95] T. McCarty. Personal Communication, 1995.

[Men64] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, Princeton, N.J., 1964.

[MSL92] D. Mitchell, B. Selman, and L. Levesque. Hard and easy distribution of SAT problems. In *Proceedings Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.

[New42] M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[Rd87] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 183–188, 1987.

[SK91] B. Selman and H. Kautz. Knowledge compilation using Horn approximations. In *Proceedings Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 904–909, 1991.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. computer science press, Rockville, MD, 1988.

[Yas94] A. Yasuhara. Logic, Computability and Complexity. Submitted for publication, 1994.