Logic Programming:
Operational Semantics and Proof Theory

James Harold Andrews

Doctor of Philosophy
University of Edinburgh
1991
Abstract

Logic programming systems which use parallel strategies for computing "and" and "or" are theoretically elegant, but systems which use sequential strategies are far more widely used and do not fit well into the traditional theory of logic programming. This thesis presents operational and proof-theoretic characterisations for systems having each of the possible combinations of parallel or sequential "and" and parallel or sequential "or".

The operational semantics are in the form of an abstract machine. The four control strategies emerge as simple variants of this machine with varying degrees of determinism; some of these variants have equivalent, compositional operational semantics, which are given.

The proof-theoretic characterisations consist of a single central sequent calculus, LKE (similar to Gentzen's sequent calculus for classical first order logic), and sets of axioms which capture the success or failure of queries in the four control strategies in a highly compositional, logical way. These proof-theoretic characterisations can be seen as logical semantics of the logic programming languages.

The proof systems can also be used in practice to prove more general properties of logic programs, although it is shown that they are unavoidably incomplete for this purpose. One aspect of this incompleteness is that it is not possible to derive all valid sequents having free variables; however, induction rules are given which can help to prove many useful sequents of this kind.
Acknowledgements

Thank you:

To the British Science and Engineering Research Council, Edinburgh University, the Edward Boyle Memorial Trust, and Bell-Northern Research Inc., for their generous financial assistance;

To my advisors, Don Sannella and Stuart Anderson, for many fruitful meetings;

To Inge-Maria Bethke, Ruth Davis, Paul Gilmore, Lars Hallnäs, James Harland, Bob Harper, Gordon Plotkin, David Pym, Peter Schroeder-Heister, and David Walker, for helpful comments and suggestions;

To Paul Voda, for his vision and for setting me on the course of research that led to this thesis;

And to Julian Bradfield, Leslie Henderson, Jane Hillston, Craig McDevitt, James McKinna, and the Salisbury Centre Men’s Group, for much-appreciated friendship and support.

This thesis is dedicated to the memory of my father, Stewart James Andrews.
Declaration

This thesis was composed by myself. The work reported herein, unless otherwise stated, is my own.

Some of the work contained in this thesis has appeared before in a different form [And89b, And89c, And90a, And90b].
# Table of Contents

1. Introduction 8

1.1 Programming Languages and Semantics 9

1.2 Logic Programming 9

1.2.1 Declarative Semantics 11

1.2.2 Operational Semantics 12

1.2.3 Declarative Semantics Revisited 13

1.3 The Approach and Scope of This Thesis 14

1.4 Definitional Preliminaries 17

2. Operational Semantics 20

2.1 Control Disciplines and their Uses 22

2.2 Stack-of-Stacks Operational Semantics: SOS 24

2.2.1 Definitions and Rules 24

2.2.2 Properties of SOS 27

2.2.3 Discussion 30

2.3 Sequential Variants of SOS 31

2.3.1 Definitions and Rules 32

2.3.2 Completeness Properties of Sequential Computations 34

2.4 Compositional Operational Semantics 38
### Table of Contents

2.4.1 One-Stack Operational Semantics OS: Parallel Or ..... 38
2.4.2 One-Stack Operational Semantics OSso: Sequential Or ... 41
2.4.3 One-Formula Operational Semantics Csa .................. 42

2.5 Some Properties of Existential Quantification ............... 45
2.5.1 Failure ........................................ 46
2.5.2 Success ....................................... 47
2.5.3 Solution Completeness .......................... 49

2.6 Summary and Classification of Queries ..................... 50

3. Characterising Parallel Systems .......................... 52
3.1 Overview and Definitions ................................ 53
3.2 LKE and Its Soundness .................................. 56
3.3 Axioms for Parallel Validity ............................ 61
3.3.1 Predicate Unfoldings and the $F^N$ sign .............. 61
3.3.2 The PAR Axioms and their Validity .................. 63

3.4 Completeness: Closed Assertions ......................... 67
3.4.1 Completeness for Equality Sequents .................. 67
3.4.2 Closed Completeness ............................. 70
3.4.3 Characterisation Results .......................... 74

3.5 Completeness: Predicate-Free Assertions .................. 75

3.6 Discussion .......................................... 78
3.6.1 LKE ............................................. 78
3.6.2 PAR ............................................ 79
3.6.3 Characterising Success Alone ....................... 80
3.6.4 More Practical Failure Rules ....................... 81
# 4. Characterising Sequential Systems

4.1 Approaches to Semantics

4.1.1 Operational Approaches

4.1.2 Denotational Approaches

4.2 Disjunctive Unfoldings

4.2.1 An Initial Attempt at a Characterisation

4.2.2 Definitions and Examples

4.2.3 Some Facts about Unfoldings

4.3 Axioms for Sequential Validity

4.4 Completeness: Closed Assertions

4.5 Completeness: Predicate-Free Assertions

4.6 Success for SOS/so

4.7 Discussion

4.7.1 SEQ as a Characterisation of SP

4.7.2 The Inversion Principle for SEQ

# 5. Approaches to Incompleteness

5.1 Incompleteness

5.1.1 The Halting Problem and Divergence

5.1.2 Gödel Incompleteness

5.2 Infinitary Methods

5.2.1 An Infinitary Rule for Free Variables

5.2.2 Model Checking for Divergence

5.3 Induction

5.3.1 Subterm Induction
## Table of Contents

5.3.2  Disadvantages of Subterm Induction ....................... 126

5.3.3  Well-Founded Induction .................................. 128

6.  Summary and Future Directions ................................. 131

6.1  Language Extensions ........................................... 132

6.1.1  Iterated Implication and Universal Quantification ....... 133

6.1.2  Types .................................................... 134

6.1.3  Variable Modes ........................................... 134

6.1.4  Predicate Names as Terms ................................ 135

6.1.5  Negation and If-Then-Else ................................ 136

6.1.6  Constraints ............................................... 138

6.1.7  Multiple Control Disciplines ............................... 139

6.2  Practical Program Proving ..................................... 139

A.  Examples ...................................................... 149

A.1  Conventions ................................................... 149

A.2  List Membership Examples .................................... 150

A.2.1  Computations .............................................. 150

A.2.2  Derivations ............................................... 151

A.3  Infinite Loop .................................................. 151

A.3.1  Derivations ............................................... 154

A.4  Subterm Induction .............................................. 154

A.5  Well-Founded Induction ....................................... 157

Index of Definitions ............................................. 161
Chapter 1

Introduction

The quest for programming languages which are more readable and expressive has led to many developments in programming languages, one of which is the logic programming paradigm. In theory, logic programming languages are more readable and expressive because they borrow some of the expressive power of the language of mathematical logic – a language which was developed specifically in order to model some of the deductive processes of the human mind.

This theoretical goal has been achieved to only a limited extent in practice, because the implementations of logic programming languages differ from the ideal theoretical model in many ways. One of the most basic and profound of the differences is that the theory concerns languages which can be implemented completely only by parallel (breadth-first) interpreters, while most practical implementations use incomplete, sequential (depth-first) strategies.

This incompleteness in itself would not necessarily be a problem; but unfortunately, the exact set of terminating sequential logic programs is hard to characterise in a logical way. Sequentiality also affects reasoning about programs, disrupting the hope that the identification of program with logical formula would make this straightforward. These problems tend to weaken claims that practical and truly logical programming is possible.

This thesis is intended as a step towards mending this rift between theory and practice, between parallel and sequential systems. In the thesis, I present a
homogeneous operational characterisation of the parallel and sequential versions of a basic logic programming language; I then use proof systems to characterise, in a logical manner, the sets of queries which terminate in the various parallel, sequential, and mixed control disciplines. I also show how these proof systems can be used to prove more general properties of logic programs.

By way of introduction, I will present some discussion about the general principles and historical development of programming languages and semantics. I then will focus on logic programming, addressing in particular the various approaches to its declarative and operational semantics, and the associated problems. Finally, I will delineate the approach and scope of this thesis, and end this introduction with some definitional preliminaries.

1.1 Programming Languages and Semantics

In the first computer languages, programs consisted of sequences of encoded instructions which described how the computer was supposed to change its internal state at each step of the computation. These languages were very "machine-oriented", in the sense that they expressed directly what the machine was to do. To write a program, programmers had to find out how to express the problem to be solved as a sequence of instructions.

Programmers soon came to realise that certain constructs corresponding to higher-level concepts were being used again and again. Compilers and interpreters were introduced in order to allow programmers to express these higher-level concepts more directly, with the compiler or interpreter handling the automatic translation into the standard constructs: languages thus became more "human-oriented". For instance, in FORTRAN, programmers were for the first time able to write arithmetic expressions directly, and expect the compiler to generate the appropriate sequence of loads, stores and arithmetic operations.

The concepts of procedures and functions, structured programming, functional, logic, and object-oriented programming all arose out of similar desires to make
high-level concepts clearer. Languages can now be grouped into various so-called "paradigms", according to how a program is viewed in the language. In imperative languages, a program is a sequence of instructions. In functional languages, it is a set of formal declarations of functions. In logic programming languages, it is a set of logical expressions acting as a "knowledge base". And in object-oriented languages, it is a description of communications between agents.

To some extent, semantics of programming languages reflect the desire to view programs at a high level. One important thing that a formal semantics can give us is an abstract mathematical account of the exact sense in which a programming language encodes higher-level concepts. Thus, functional programming language semantics associate function definitions with actual mathematical functions; logic programming language semantics associate programs with models, which in turn are characterisations of "possible states of knowledge". A good semantics describes rigorously how we expect programs to behave at a high level, and does so in terms of the intended paradigm.

However, in addition to a high-level view, we still need descriptions of programming languages which capture what is going on at the basic level within the machine. We need such descriptions in order to tell how much time and space our programs are going to take up and how we can improve our programs' efficiency, and in order to be able to follow the execution of our programs for debugging and testing purposes.

These computational considerations are somewhat in conflict with the other purposes of semantics. The way in which we usually resolve this conflict is to give one "operational" semantics which meets our computational criteria, and one or more other semantics which give a higher-level, "declarative" view of the meanings of programs. We then formally prove the equivalence of the two semantic descriptions, to allow us to get the benefits of both.
1.2 Logic Programming

In logic programming, in particular, the tradition since the earliest papers in semantics [vEK76] has been to give an operational (or "procedural") semantics (usually based on SLD-resolution [Hil74]), and one or more logical (or "declarative") semantics which give intuitive descriptions of the meanings of programs. The operational semantics, however, is usually implemented only incompletely, so if we want to keep the correspondence between operational and declarative semantics we must find new declarative semantics which correspond to the incomplete implementations.

1.2.1 Declarative Semantics

The declarative semantics of logic programming is a reflection of what we want to use logic programming for. Some programming tasks have an inherently logical nature, such as knowledge-based systems and automated theorem provers. Other tasks, such as protocol testers and definite clause grammar parsers, are not immediately logical but are easily expressed in logic. The language of formal logic was developed specifically in order to express such ideas symbolically, particularly in the realm of mathematics; its semantics, model theory, is based on mathematical structures which reflect notions of a priori and derived truth. Thus, when we write a logic program, we have (if we understand logic and its model theory) a clear intuitive idea of what it means.

The traditional approach to logic programming semantics [vEK76,Llo84,Fit85] views a program as a set of logical formulae (Horn clauses). The "denotation" of the program is the least model of this set of formulae; a query to the program is satisfiable iff some closed instance of it is in the least model. It is the intention that the operational semantics, when given a program and a query, finds a satisfying substitution to the query if one exists. In the traditional approach, there is also a fixpoint semantics which acts as a "bridge" between the operational and declara-
tive semantics, for the purpose of proving soundness and completeness properties of the operational semantics.

There is also a more minor tradition of proof-theoretic characterisation of logic programs [HS84, HSH88, And89b]. One goal of this approach is to combine the logicalness of model-theoretic descriptions with the precision and formality of operational descriptions. A proof-theoretic characterisation associates a program with a proof system with respect to which the operational semantics is sound and complete. The soundness and completeness typically takes the following form: a query is successful iff an associated formula is a theorem of the proof system.

Proof-theoretic characterisations have several advantages. As in least model semantics, programs are described in terms of the logic underlying our intuitions about programs. But unlike model-theoretic approaches, the characterisation is purely formal and syntactic, so the proofs of soundness and completeness can be purely syntactic, with little reference to abstract notions. If the characterisations are in the form of natural deduction proof systems, they can be as clear and understandable as such proof systems are for describing conventional logics (see Gentzen [Gen69] or Sundholm [Sun83]).

Finally, characterising proof systems can be used directly to build systems for verifying correctness and termination properties of logic programs [And89b]. Schemes for proving properties of programs [BC89, Der89] are related to the proof-theoretic tradition; they may be seen as "axiomatic" semantics (to use Stoy's [Sto77] terminology) for logic programming.

1.2.2 Operational Semantics

The innovations of the logic programming paradigm lie to a large extent in its operational semantics. From a purely algorithmic viewpoint, a major advantage of logic programming for certain tasks is that the description of the solution can be expressed directly, while details of the control of search for the solution do not appear explicitly in the program. This is not to say that programmers do not need to be aware of the search strategy, however; if they are to write programs which
are efficient and correct, they must know how the computer executes them. This is where the operational semantics can help.

Traditionally, operational semantics are given by some variant of SLD-resolution, which is in turn a specialisation of Robinson's resolution strategy [Rob65]. SLD-resolution is sound and complete with respect to the model theory: whenever a query is satisfiable, it will find a solution, and it will never give a wrong solution. But SLD-resolution is nondeterministic in the sense that there may be more than one direction to proceed to find a proof of a given query. So to implement SLD-resolution completely, we need some kind of breadth-first search of the search space; but breadth-first search can be very inefficient, and for a number of reasons which I will discuss later, very few practical systems implement it.

The approach of most implementers of Prolog and other logic programming languages has been to implement SLD-resolution only partially and incompletely, or even unsoundly. The "occurs check" in the unification algorithm is often done away with, even though this results in occasional wrong solutions; and a sequential, left-to-right, depth-first search rule is often used to search the solution space, even though this does not find all solutions.

This sequential strategy has a fairly simple operational semantics, easily characterisable within the framework of SLD-resolution. But because it is not complete, there are queries which succeed, or fail, in full SLD-resolution which diverge (fail to terminate either way) in the sequential version. The loss of completeness is not felt very acutely by programmers, but the theoretical tradition has continued to study mostly the nondeterministic SLD-resolution and its model theory.

1.2.3 Declarative Semantics Revisited

How does this incompleteness of sequential Prolog reflect upon the declarative semantics of logic programming? The converse of this incompleteness is that the least-model and fixpoint semantics do not characterise sequential Prolog precisely: the set of successful queries in sequential Prolog is a subset of the set of satisfiable queries in the least model semantics. So there seems to be a split between theory
and practice in logic programming, in which the theory is mostly about parallel systems and the practice mostly about sequential systems. To regain the original goals of the logic programming paradigm – practical programming languages based on logic – it seems that we have to either abandon the practical systems or to give an abstract, declarative semantics for sequential Prolog.

The latter alternative seems to be desirable, but very little work has been done in this area. What has been done consists mostly of denotational semantics [JM84, Bau88,NF89]. Denotational semantics characterises well the set of successful and failing queries, but it does so in a functional, non-logical manner. This is discouraging, because if we take a semantics as being a reflection of the intended paradigm, it suggests that practical logic programming is not “logical” at all, but a logical-looking variant of functional programming.

The main goal of this thesis is to give a declarative semantics of sequential logic programming, but to do so in a way which shows the language’s clear connections with logic. In doing so, I hope to achieve a view of logic programming in which both the parallel and the sequential versions of the language fit equally well into an overall operational and declarative framework.

1.3 The Approach and Scope of This Thesis

This thesis follows the usual pattern of giving an operational semantics and a logical characterisation for logic programming languages, and proving equivalences between them.

I concentrate on studying the behaviour of the logical connectives in various logic programming systems, and in particular the behaviour of “and” and “or”. I am specifically not concerned with other issues of incompleteness or unsoundness in logic programming implementations, such as the occurs check. I deal with a very simple language equivalent to Horn clauses – one which has no negation or cut, for instance – although I do suggest ways in which the language might be extended.
In the first technical chapter, Chapter 2, I give a definition of a very general operational semantics for this logic programming language. The operational semantics, SOS, is in the form of a formal tree-rewriting system or formal abstract machine. The basic operational semantics is one for a parallel-and, parallel-or system; a simple restriction on the application of the transition rules leads to a sequential-and system, and another, similar restriction leads to a sequential-or system. (Both restrictions together give the common "sequential Prolog" system.) I use SOS rather than the traditional SLD-resolution because it allows us to describe these various control strategies, including the failure-and-backtrack behaviour of sequential Prolog, within the formal system.

In Chapter 2, I also give several more compositional variants of SOS, which allow us to see more clearly some of the higher-level properties of computation in SOS. The chapter ends with a classification of the queries which succeed and fail with the four major control strategies (sequential or parallel "and", sequential or parallel "or").

In Chapters 3 and 4, I present the declarative semantics, which follow the proof-theoretic tradition mentioned above. The semantics take the form of sequent calculi. The elements of sequents in the calculus are assertions, which are expressions built up from signed formulae by the connectives of classical logic. Signed formulae, in turn, are essentially logic programming goal formulae enclosed by the sign $S$ (for success) or $F$ (for failure). The sequent calculi therefore retain a strong logical flavour.

The calculi in the two chapters share a set of rules called LKE. LKE is basically a Gentzen-style sequent calculus for classical first order logic with equality as syntactic identity. In Chapter 3, I concentrate on the problem of characterising queries which fail in parallel-and systems, and those which succeed in parallel-or systems; LKE is augmented by a set of simple logical axioms which describe the success and failure behaviour of queries under these assumptions. I prove the soundness of all the rules, and various completeness results about useful classes of sequents. One such completeness result is that if a query $A$ succeeds (resp. fails), the sequent $\rightarrow S(\exists[A])$ (resp. $\rightarrow F(\exists[A])$) is derivable, where $\exists[A]$ is the
existential closure of $A$. This amounts to a precise and logical characterisation of the sets of successful and failing queries.

In Chapter 4, on the other hand, I concentrate on characterising queries which succeed or fail in sequential-and, sequential-or systems. There is a similar set of axioms which correspond to this assumption; these axioms are made simple and compositional by the introduction of the notion of disjunctive unfolding of a formula. I prove similar soundness and completeness results about this calculus.

In addition to being able to prove assertions about the success and failure of individual queries, the sequent calculi are able to prove much more general properties of programs – such as the disjunctions, negations, and universal generalisations of such assertions. They therefore have important practical applications in software engineering: they can act as a basis for practical systems for proving properties of logic programs, such as proving that a program meets its specification.

There are limitations to how complete a finitary sequent calculus can be for this purpose, however. In Chapter 5, I explore this question of incompleteness. I give incompleteness results based on the Halting Problem and Gödel's incompleteness theorem to define some of the boundaries of the proof-theoretic approach. However, I also give some ways in which we can partially deal with this incompleteness: infinitary rules (a standard technique from logics of programs) and induction rules (which are very useful for practical program proving).

Finally, I discuss the import of the results and the directions that future research in this area could take in Chapter 6, and I give an Appendix with some examples of computations and derivations. There is an Index of Definitions at the end of the thesis which contains entries for all the formal, numbered definitions.
1.4 Definitional Preliminaries

Definition 1.4.1 A *first-order language* $\mathcal{L}$ consists of a set $\mathcal{X}(\mathcal{L})$ of variable names, a finite set $\mathcal{F}(\mathcal{L})$ of function symbols $f_i$ each with an associated arity $n_i \geq 0$, and a set $\mathcal{P}(\mathcal{L})$ of predicate names $P_j$ each with an associated arity $m_j \geq 0$.

The *terms* of a language $\mathcal{L}$ are inductively defined as follows: every variable of $\mathcal{L}$ is a term, and every expression $f(t_1, \ldots, t_n)$, where $f$ is a function symbol of arity $n$ and $t_1, \ldots, t_n$ are terms, is a term; nothing else is a term. Nullary function symbol applications $f()$ are often written as simply $f$.

Following Miller and Nadathur [MN86], I will define the class of "goal formulae" as a restricted class of first order formulae built up from predicate applications using only the connectives "and", "or", and "there exists".

Definition 1.4.2 A *goal formula* in a first-order language $\mathcal{L}$ with a binary equality predicate $=$ is an expression which meets the following BNF syntax:

$$
G ::= s = t \mid P(t_1, \ldots, t_n) \mid G_1 \& G_2 \mid G_1 \lor G_2 \mid \exists x \ G
$$

where $s$, $t$, and the $t_i$'s are all terms of $\mathcal{L}$, $P$ is a predicate name of $\mathcal{L}$ of arity $n$, and $x$ is a variable name of $\mathcal{L}$.

The class of goal formulae of $\mathcal{L}$ is thus a subclass of the class of formulae of $\mathcal{L}$. I will treat the word "query" as a synonym for "goal formula"; but I will use the former when we want to refer to a formula, possibly having free variables, for which we ask a logic programming system to find a satisfying substitution.

In the traditional approach to logic programming, used to describe Prolog and similar languages, programs are defined as sets of Horn clauses. Because I wish to compare logic programming systems directly with proof systems, I adopt a form of predicate definition which looks more like the completion of a predicate [Cla78].
Definition 1.4.3 A predicate definition in \( \mathcal{L} \) (defining the predicate \( P \)) is an expression of the form
\[
P(x_1, x_2, \ldots, x_n) \leftrightarrow A
\]
where the left- and right-hand sides of the \( \leftrightarrow \) are goal formulae in \( \mathcal{L} \), the \( x_i \)'s are distinct, and the right-hand side contains no free variables not in \( \{x_1, x_2, \ldots, x_n\} \).

Definition 1.4.4 A program in \( \mathcal{L} \) is a finite set of predicate definitions in which all names of predicates being defined are distinct.

This form is no loss or gain of power over the clausal form, but it makes connectives explicit and allows us to examine their effect and processing directly. To get a program of this form from a Horn clause program, we need only take the completion of the program [Cla78]. Example: in a language with a binary function symbol \([\cdot,\cdot]\) of list formation, the standard "member" predicate \( \text{Mem} \) might be defined as follows:
\[
\text{Mem}(x, l) \leftrightarrow \exists h \exists t \ (l = [h|t] \& (x = h \lor \text{Mem}(x, t)))
\]

We will generally want to interpret a query \( A \) given to a logic programming system as a request to prove the existential closure of \( A \). Since the logic we use will turn out to be constructive, this is the same as a request to find a satisfying substitution for \( A \); the existential-closure view will just facilitate the comparison with the proof theory.

Definition 1.4.5 The existential closure of a formula \( A \), in symbols \( \exists[A] \), is the formula \( \exists x_1 \ldots \exists x_n A \), where \( x_1 \ldots x_n \) are all the free variables of \( A \).

In the sequel we will assume the existence of some fixed first-order language \( \mathcal{L} \) with equality \( = \) as the language of all programs. We will further assume that \( \mathcal{L} \) generates at least two closed terms. (This is not really a restriction for most practical applications.) We will write these closed terms as \( 0 \) and \( 1 \), and define the formula \( \text{true} \) as an abbreviation for \( 0 = 0 \), and \( \text{false} \) as an abbreviation for \( 0 = 1 \).
In most of what follows, we will also assume the existence of at least one non-nullary operation symbol; that is, we will assume that the language has an infinite number of closed terms. This is a reasonable assumption for most logic programming applications, but not so for the area of logic databases, where there may be only a finite number of data elements in the domain of discourse. I will point out the use of the assumption of an infinite domain whenever it is used, and discuss the implications of this.

Other notation is as follows. A, B, C, D are metavariables standing for goal formulae; P, Q, and R stand for predicate names; r, s, and t stand for terms; and x, y, and z stand for variables.

θ and ρ stand for substitutions. Aθ stands for the application of θ to A (where all substitutions of terms for variables take place simultaneously, and may involve renaming to avoid capture of free variables); similarly tθ stands for θ applied to the term t. [x := t] is the substitution which maps only x to t.

I use the notation A(s) and then later A(t) to mean A[x := s] and then later A[x := t], for some given formula A with some given variable x free. (A(x) should not be confused with P(x), which is an application of predicate P to variable x.) Similarly I use r[s := t], r(s), r(t).
Chapter 2

Operational Semantics

Operational (or "procedural") semantics, as I mentioned in the Introduction, are used to provide characterisations of programming languages which meet certain "computational" criteria: giving a detailed description of the language for implementation purposes, and giving a computational model to which programmers can refer.

For logic programming, operational semantics are particularly important because it is in them that the innovations of logic programming lie. The notions of resolution and unification are not immediately apparent; unification, though defined by Herbrand in his thesis [Her30], was virtually ignored until Prawitz's work [Pra60], and resolution was not defined until 1965 [Rob65]. These notions must be explained within the context of a full description of the computational model of the language.

If we want to do such things as soundness and completeness proofs, or indeed any formal comparison of the operational semantics to other characterisations of the language, the operational semantics must also be mathematically precise – for instance, in the form of a formal system. (Plotkin [Plo81] has explored the idea of structured operational semantics in detail, and gives a taxonomy to which I will refer in this chapter.) SLD-resolution [Kow74], SLDNF-resolution [Llo84], and the operational semantics in this chapter are just a few examples of formal operational semantics for logic programming. Other examples include Voda's tree-rewriting system [Vod85], Deransart and Ferrand's [DF87] and Börger's [Bör90]
standardisation efforts, and the abstract computation engines for Andorra Prolog [HJ90] and the "Pure Logic Language", PLL [Bab91, McB91]. The operational semantics which is closest to the one in this chapter is perhaps that in de Bruin and Vink's continuation semantics paper [dBdV89].

SLD-resolution is an adequate operational semantics for logic programming, but one not quite suited to our purposes. SLD-resolution is centred around the idea of a program as a set or sequence of Horn clauses, which in turn are sets or sequences of literals. In our view of a program as a set of definitions of predicates by goal-formula bodies, we must give an operational semantics which manipulates goal formulae as its basic units. In the various operational semantics to be given in this chapter, each transition rule concerns itself with a single form of goal formula (disjunction, conjunction, predicate call and so on), in keeping with our study of how systems treat the various connectives. In addition, the nature of SLD-resolution requires us to use rules external to the operational semantics ("subgoal selection rules" and "search rules") to characterise such things as backtracking in sequential-or systems; such notions are entirely integrated into the operational semantics in this chapter.

In this chapter, I first discuss the various control disciplines and their uses, and then give the parallel-and, parallel-or operational semantics SOS ("stack of stacks"), which will act as a reference point throughout the rest of the thesis. Sequential-and and sequential-or systems are simple variants of SOS, and have certain completeness properties with respect to SOS; these I describe next. I also give some more compositional variants of SOS, which will be useful in later chapters for proving theorems. I give some somewhat technical lemmas about the behaviour of existential quantifiers, which will be used frequently in later chapters. The chapter ends with a summary of the classification of all queries according as if they succeed, fail, or diverge in the various control disciplines described in this chapter.
2.1 Control Disciplines and their Uses

This chapter gives operational semantics for several different control disciplines. It will therefore be useful to review why we want to do this, and what the various control disciplines are useful for.

The implementation of a logic programming system involves choices about how complete it is going to be. When we give a query formula $A$ with free variables to the system, we want it to find a substitution $\theta$ such that $A\theta$ is true, given the standard definition of logical truth and the predicate definitions in the program. The ideal — a complete system — is a system which can find such substitution whenever it exists.

Unfortunately, this ideal cannot be realised without there being some form of parallelism in the interpreter — that is, some coprocessing, timesharing, or breadth-first search. The crucial problems centre around the two binary connectives, the conjunction $\&$ and the disjunction $\lor$. Say that the interpreter processes queries of the form $B\&C$ by first processing $B$ and then processing $C$. Then, although a query such as $B\&false$ has no solution, if the processing of $B$ causes the interpreter to go into an infinite loop, then the interpreter will never discover that it has no solution: the interpreter will be incomplete for failure. Similarly, interpreters which process disjunctions in this sequential manner will be incomplete for success, never finding solutions to some queries which have solutions.

Nevertheless, this “sequential” control discipline for one or both of the binary connectives is very common in many logic programming systems, and is likely to remain so [Dev90]. Sequential “and” and “or” are easy to implement in a simple stack-based interpreter, and yield an execution model, based on sequential execution and backtrack, which makes it relatively easy to follow the execution of a program for debugging purposes. In contrast, giving an efficient implementation of a language with parallel “and” and “or” involves optimising such things as resource sharing and process switching. It is also more difficult to debug a program when there are several things going on at once, as in parallel programs, rather than
just one stream of computation. Added to these technical considerations is the empirical observation that the loss of completeness in sequential systems appears to be relatively easy for programmers to work around.

Sequential "and" and "or" are, therefore, both used in most commercial Prologs and other logic programming languages. Parallel "and" and "or" [CM83,CG83, Sha89] are the exception, rather than the rule, and are generally used only when there is some particular application that works better with them. Parallel "or" is useful for problems with a complex logical structure, in which it is unclear which of several alternatives will lead to a solution. Since parallel-or systems are complete for successful queries (as we shall see below), one use of such systems might be as execution engines for executing Horn clause specifications, because such specifications are intended to be very high-level and free from operational considerations.

Parallel "and" systems are useful for problems which can benefit from the more traditional parallelisation of computation, done to make independent computations more efficient. They are also useful for doing programming in which predicate calls joined by a conjunction represent communicating agents, such as sequential processes, coroutines, or objects [ST83,Sha89].

In some systems with parallel connectives [Vod85], the sequential versions of these connectives are also available (usually distinguished by a different syntax). Here, I will be simplifying the situation and considering only systems in which there is one kind of each connective; there is one conjunction, which is either always parallel or always sequential, and similarly one disjunction. I do this mainly to make the operational semantics simpler and thus more amenable to analysis.
2.2 Stack-of-Stacks Operational Semantics: SOS

The first operational semantics I will describe will be a formalisation of the common "stack of stacks" implementation of Prolog. It is called the stack of stacks technique because the machine state is given by a "backtrack stack" whose elements are "closures" containing "goal stacks".

In Plotkin's taxonomy [Plo81], this would be a "multi-step" system; the rules for the formal system define one step in a computation, and we are more interested in the transitive closure of the relation thus defined. Later we will see "single-step" systems, in which the relation is between one state and the final state of the entire computation.

I use the word "stack" to emphasise the analogy of certain expressions with the stack data structure known to computer programmers, because it is this data structure that is used in many implementations. However, such implementations are sequential and deterministic, and we will be studying operational semantics which may be parallel and nondeterministic. Thus, something which is called a stack here might not always be operated on by the usual push, pop, and top operations on stacks, but also by the selection of an arbitrary element from the stack, or the replacement of an arbitrary element by 0 or more elements. I hope that the advantage of familiar terminology will outweigh the disadvantage of imprecise usage. I will point out the cases in which a stack really is behaving as a computer science stack.

2.2.1 Definitions and Rules

Definition 2.2.1 A goal stack is just a sequence of formulae. We will generally use \( \alpha \), possibly subscripted, to stand for an arbitrary goal stack. We will write the concatenation of two goal stacks \( \alpha_1 \) and \( \alpha_2 \) as \( \alpha_1, \alpha_2 \). We will sometimes refer to the formulae in a goal stack as subgoals.
A closure is an expression \((\theta : \alpha)\), where \(\alpha\) is a goal stack and \(\theta\) is a substitution. (In practice, \(\theta\) will be some syntactic representation of a substitution on a finite number of variables.)

A backtrack stack is a sequence of closures. We will generally use \(\beta\), possibly subscripted, to stand for an arbitrary backtrack stack. We will write the concatenation of two backtrack stacks \(\beta_1\) and \(\beta_2\) as \(\beta_1 \beta_2\).

\(\epsilon\) will represent the empty goal stack or backtrack stack (its use will be non-ambiguous).

Logically, a backtrack stack
\[
(\theta_1 : A_{\alpha_1}; \ldots; A_{\alpha_{n_1}}); \ldots; (\theta_m : A_{\alpha_{m_1}}; \ldots; A_{\alpha_{m_m}})
\]
represents the formula
\[
(A_{\alpha_{1_1}} \theta_1 \& \ldots \& A_{\alpha_{1_{n_1}}} \theta_1 \& \text{true}) \lor \ldots \lor (A_{\alpha_{m_1}} \theta_m \& \ldots \& A_{\alpha_{m_m}} \theta_m \& \text{true}) \lor \text{false}
\]
for which we are trying to find a satisfying substitution. Thus the elements of a goal stack represent subgoals, all of which are to be solved in the context of the substitution in their closure; and the elements of a backtrack stack represent different alternatives, any one of which may yield a solution.

There will be one operational semantics, called \(\text{SOS}_\Pi\), corresponding to each program \(\Pi\). (Since the particular program being considered will usually not be important, we will usually drop the subscript.) \(\text{SOS}\) is a rewriting system which consists of rules for a binary relation \(\Rightarrow^\text{SOS}\) between backtrack stacks, and a definition of which backtrack stacks are to be considered “success states” and “failure states”.

The rewriting rules of \(\text{SOS}\) are in Figure 2-1. The success states of \(\text{SOS}\) are all backtrack stacks of the form
\[
\beta_1; (\theta : \epsilon); \beta_2
\]
that is, all backtrack stacks containing a closure with an empty goal stack. The single failure state of \(\text{SOS}\) is \(\epsilon\), the empty backtrack stack.
1. &:
\[ \beta_1; (\theta : \alpha_1, (B & C), \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; (\theta : \alpha_1, B, \alpha_2); \beta_2 \]

2. v:
\[ \beta_1; (\theta : \alpha_1, (B \lor C), \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; (\theta : \alpha_1, B, \alpha_2); (\theta : \alpha_1, C, \alpha_2); \beta_2 \]

3. \exists:
\[ \beta_1; (\theta : \alpha_1, (\exists x B), \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; (\theta : \alpha_1, B[x := x'], \alpha_2); \beta_2 \]
where \( x' \) is some variable not appearing free to the left of the arrow

4. Defined predicates:
\[ \beta_1; (\theta : \alpha_1, P(t_1, \ldots, t_n), \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; (\theta : \alpha_1, A(t_1, \ldots, t_n), \alpha_2); \beta_2 \]
where \( \Pi \) contains the definition \((P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n))\)

5. Unification, success:
\[ \beta_1; (\theta : \alpha_1, s = t, \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; (\theta \theta' : \alpha_1, \alpha_2); \beta_2 \]
where \( \theta' \) is the mgu of \( s \theta \) and \( t \theta \)

6. Unification, failure:
\[ \beta_1; (\theta : \alpha_1, s = t, \alpha_2); \beta_2 \xrightarrow{\text{SOS}} \beta_1; \beta_2 \]
where \( s \theta \) and \( t \theta \) do not unify

Figure 2-1: Rules for the operational semantics SOS.
Chapter 2. Operational Semantics

To execute a particular goal formula $A$ in this interpreter, we form the backtrack stack consisting of the single closure $((():A))$, where $()$ is the empty substitution. We then repeatedly apply appropriate transitions. If we reach a failure state, we conclude that the query is unsatisfiable; if we reach a success state $\beta_1; (\theta : \epsilon); \beta_2$, we conclude that $\theta$ is a satisfying substitution for $A$. Of course, for a given goal we might never reach a success or a failure state, due to repeated diverging applications of the defined predicate rule.

Definition 2.2.2 The relation $\Rightarrow^*$ is the reflexive and transitive closure of the $\Rightarrow$ relation.

We say that a backtrack stack $\beta$ succeeds if $\beta \Rightarrow^* \beta'$, where $\beta'$ is a success state.

We say that $\beta$ fails if $\beta \Rightarrow^* \epsilon$.

We say that $\beta$ diverges if it neither succeeds nor fails.

We say that a query formula $A$ succeeds (fails, diverges) if the backtrack stack $((():A))$ succeeds (fails, diverges), where $()$ is the empty substitution.

We say that SOS describes a "parallel and" discipline because each conjunction in a backtrack stack is processed by splitting it into its subformulae, after which point the computational rules can operate on either subformula. We say that it describes a "parallel or" discipline for similar reasons.

2.2.2 Properties of SOS

We will leave discussion of the logical properties of this operational semantics to later chapters, and here will concentrate on the combinatorial, operational properties of SOS. To begin, SOS has the Church-Rosser property that any two computations diverging from the same point can converge again (up to a natural equivalence).
We must begin by proving the Church-Rosser property of a slightly weaker system, namely the system SOS without the Defined Predicate rule. The notion of equivalence is the same.

**Definition 2.2.3** Two backtrack stacks $\beta_1, \beta_2$ are *renaming-permutation-equivalent* if they are identical up to a (closure-wise) renaming of variables and a permutation of closures.

Let $f\text{SOS}$ be the operational semantics $\text{SOS}$ without the Defined Predicate rule.

**Theorem 2.2.4 (Church-Rosser Property of $f\text{SOS}$)**

1. (Diamond property.) If $\beta \xrightarrow{f\text{SOS}} \beta_1$ and $\beta \xrightarrow{f\text{SOS}} \beta_2$, then there is a $\beta'$ such that $\beta \xrightarrow{f\text{SOS}} \beta_1$ and $\beta \xrightarrow{f\text{SOS}} \beta_2$, up to renaming-permutation equivalence.
2. (Strong normalisation.) There is no infinite computation sequence in $f\text{SOS}$, and every computation sequence can be extended to reach a backtrack stack for which no further transitions are possible.
3. (Church-Rosser property.) If $\beta \xrightarrow{f\text{SOS}^*} \beta_1$ and $\beta \xrightarrow{f\text{SOS}^*} \beta_2$, then there is a $\beta'$ such that $\beta_1 \xrightarrow{f\text{SOS}^*} \beta'$ and $\beta_2 \xrightarrow{f\text{SOS}^*} \beta'$.

**Proof.** (1.) By case analysis. If the two computations select the same subgoal in the same closure, then $\beta_1$ and $\beta_2$ are already renaming-permutation-equivalent. Otherwise, a different subgoal, possibly in different closures, has been selected for processing. The cases are on the form of the subgoals selected.

If both subgoals are existential formulae, then the computations can converge in one step. If both computations selected the same new variable, then the convergence is only up to a renaming of the new variables.

Otherwise, if each subgoal is in a different closure, then the computations can converge identically in one step, because nothing that goes on in one closure can affect others.

Otherwise, both subgoals are in the same closure, but are not both existentials. If both are disjunctions, then the computations can converge in two steps, by each branch performing a Disjunction step on the two copies of the other disjunction.
The convergence here is only up to a permutation of the closures: if the two disjunctions were \( A \lor B \) and \( C \lor D \), then \( \beta_1 \) will contain closures containing (in order) \( A \ldots C, A \ldots D, B \ldots C, \) and \( B \ldots D \), whereas \( \beta_2 \) will contain closures containing (in order) \( A \ldots C, B \ldots C, A \ldots D, \) and \( B \ldots D \).

Otherwise, if both subgoals are equalities, then we can consider without loss of generality the case where the first computation is \( (\theta : s_1 = t_1, s_2 = t_2) \xrightarrow{\text{SOS}} (\theta \theta_1 : s_2 = t_2) \) and the second computation is \( (\theta : s_1 = t_1, s_2 = t_2) \xrightarrow{\text{SOS}} (\theta \theta_2 : s_1 = t_1) \).

There are two subcases.

- There is no \( \theta'_1 \) such that \( s_2 \theta \theta_1 \theta'_1 = t_2 \theta \theta_1 \theta'_1 \). In this case the first computation ends in failure. Now, if there were a \( \theta'_2 \) such that \( s_1 \theta \theta_2 \theta'_2 = t_1 \theta \theta_2 \theta'_2 \), then (by the properties of mgu on \( \theta_1 \), the mgu of \( s_1 \theta \) and \( t_1 \theta \) there would be a \( \theta'_1 \) such that \( \theta \theta_2 \theta'_2 = \theta \theta_1 \theta'_1 \). But since \( s_2 \theta \theta_2 \theta'_2 = t_2 \theta \theta_2 \theta'_2 \), we have that \( s_2 \theta \theta_1 \theta'_1 = t_2 \theta \theta_1 \theta'_1 \); so \( \theta'_1 \) would have the property that we started out by assuming that no substitution could have. There is therefore no such \( \theta'_2 \), and the second computation fails as well.

- There is a \( \theta'_1 \) such that \( s_2 \theta \theta_1 \theta'_1 = t_2 \theta \theta_1 \theta'_1 \). In this case (by the properties of mgu on \( \theta_2 \)), there is also a \( \theta'_2 \) such that \( \theta \theta_1 \theta'_1 \equiv \theta \theta_2 \theta'_2 \); therefore the second computation performs a Unification step, resulting in an identical goal stack (up to renaming of variables, since mgu's are unique up to renaming of variables).

Otherwise, both subgoals are in the same closure, but are not both existentials, disjunctions, or equalities. If one of the subgoals is a disjunction, then the computations can converge in two steps on the disjunction branch (similar steps on the two copies of the other subgoal), and one step on the other branch.

In all other cases (one or both subgoals are conjunctions, or one is an equality and the other is an existential), the computation of each subgoal does not interfere with that of the other, so the computations can converge in one step.

(2.) Let the ordinal measure \( m(C) \) of a closure \( C \) be \( j \cdot 2^k \), where \( j \) is the number of connectives and equality formulae in \( C \), and \( k \) is the number of disjunctions in
C. Let the measure $m(\beta)$ of a backtrack stack $\beta$ be the sum of the measures of its closures. Then every transition in fSOS lowers the measure of the backtrack stack, since every transition eliminates at least one connective or equality formula except the $V$ rule, which changes the measure from $j \cdot 2^k$ to $2((j - 1) \cdot 2^{k-1}) = (j - 1) \cdot 2^k$.

Thus, by induction, no infinite computation sequence is possible. The only backtrack stacks in which no transitions are possible are ones containing only predicate calls (ones with measure 0). We say that these backtrack stacks are in normal form. Since fSOS computation steps can be performed on any backtrack stack not in normal form, every computation can be extended to reach a backtrack stack in normal form.

(3.) (A variant of Newman’s Lemma [Bar84].) From (1) and (2). Call a backtrack stack from which two distinct normal forms are computable ambiguous. We will prove by reductio ad absurdum that there are no ambiguous backtrack stacks.

Assume that there is an ambiguous backtrack stack $\beta$, of measure $m$; in other words, that $\beta \rightarrow^\text{fSOS} \beta_1$, $\beta \rightarrow^\text{fSOS} \beta_2$, and $\beta_1$ and $\beta_2$ are in normal form and not renaming-permutation equivalent. These computations must consist of at least one step each. If the first step in the two computations is the same, then there is an ambiguous backtrack stack with measure less than $m$ (namely, the backtrack stack arrived at after this first step).

Otherwise, the two computations make a first step to $\beta_1'$ and $\beta_2'$, respectively; by the Diamond property, there are computations leading from both $\beta_1'$ and $\beta_2'$ to some $\beta'$. Now say that there is a computation leading from $\beta'$ to some normal form $\beta''$. Either $\beta''$ is equivalent to $\beta_1$, in which case it is not equivalent to $\beta_2$, in which case $\beta_1'$ is ambiguous; or $\beta''$ is equivalent to $\beta_2$, in which case it is not equivalent to $\beta_1$, in which case $\beta_2'$ is ambiguous; or $\beta''$ is equivalent to neither, in which case both $\beta_1'$ and $\beta_2'$ are ambiguous. In any case, there is an ambiguous backtrack stack of lower measure than $\beta$. By induction, therefore, there is an ambiguous backtrack stack of measure 0; but this is impossible, since backtrack stacks of measure 0 are in normal form.
Thus, the normal form of a backtrack stack $\beta$ is unique; every computation proceeding from $\beta$ can be extended to this unique normal form; and thus if $\beta \overset{\text{fSOS}*}{\Rightarrow} \beta_1$ and $\beta \overset{\text{fSOS}*}{\Rightarrow} \beta_2$, then there is a $\beta'$ (namely, the unique normal form) such that $\beta \overset{\text{SOS}*}{\Rightarrow} \beta_1$ and $\beta \overset{\text{SOS}*}{\Rightarrow} \beta_2$.

$\square$

To obtain the analogous result about SOS, we need to separate the steps of an SOS computation into the predicate "unfoldings" and the connective steps. The following notion, which will also be necessary in later proof-theoretic characterisations, will help to capture this.

**Definition 2.2.5** $\beta'$ is a **predicate 1-unfolding** of a backtrack stack $\beta$ if it is $\beta$ with one predicate application subformula $P(t_1, \ldots, t_n)$ replaced by $A(t_1, \ldots, t_n)$, where $P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n)$ is in the program $\Pi$.

$\beta'$ is a **predicate unfolding** of a backtrack stack $\beta$ if it is $\beta$ with 0 or more successive predicate 1-unfoldings applied to it.

**Theorem 2.2.6 (Church-Rosser Property of SOS)** If $\beta \overset{\text{SOS}*}{\Rightarrow} \beta_1$ and $\beta \overset{\text{SOS}*}{\Rightarrow} \beta_2$, then there is a $\beta'$ such that $\beta_1 \overset{\text{SOS}*}{\Rightarrow} \beta'$ and $\beta_2 \overset{\text{SOS}*}{\Rightarrow} \beta'$.

**Proof.** First of all, note that there are predicate unfoldings $\beta^+_{1,2}$ of $\beta$, $\beta^+_1$ of $\beta_1$, and $\beta^+_2$ of $\beta_2$, such that $\beta^+ \overset{\text{fSOS}*}{\Rightarrow} \beta^+_1$ and $\beta^+ \overset{\text{fSOS}*}{\Rightarrow} \beta^+_2$: we can identify in $\beta$ the predicate application subformula which gets unfolded first in the computation of $\beta_1$ and perform a predicate 1-unfolding on $\beta$ for that, and so on for the other predicate steps in the computations of $\beta_1$ and $\beta_2$. The fSOS computations will then be just the steps in the SOS computations other than the predicate steps; the unfoldings will not interfere with these steps. $\beta^+_1$ might not be identical to $\beta_1$ because it will have some predicates unfolded to account for the steps toward $\beta_2$, or some unfolded subformulae duplicated due to the V rule.

Now, from the Strong Normalisation and Church-Rosser properties of fSOS, we know that there is a $\beta'$ in normal form such that $\beta^+_1 \overset{\text{fSOS}*}{\Rightarrow} \beta'$ and $\beta^+_2 \overset{\text{fSOS}*}{\Rightarrow} \beta'$. $\beta'$ is thus accessible from $\beta_1$ and $\beta_2$ by a series of predicate unfoldings followed by
a series of fSOS steps. However, we can form an SOS-computation of $\beta'$ from $\beta_1$ by inserting Defined Predicate steps corresponding to each predicate unfolding, at the points where each instance of the predicate call in question becomes a top-level element of a backtrack stack. We can do the same to derive an SOS-computation of $\beta'$ from $\beta_2$.

This theorem has some interesting consequences. The most important of these are the following.

**Corollary 2.2.7** If any computation of $\beta$ reaches the failure state, then no computation of $\beta$ can reach a success state.

**Proof.** By the theorem, if this were possible, then the two computations could be extended to meet (up to renaming-permutation equivalence). But there is no computation proceeding from the failure state; and no computation proceeding from a success state (which has one closure to which no rules apply) can reach a failure state (which has no closures).

This means that a query cannot both succeed and fail; the set of succeeding backtrack stacks and the set of failing backtrack stacks are disjoint. (Below, we will study further some of the structure of these two sets.)

**Corollary 2.2.8** If a computation of $\beta$ reaches a success state $\beta'$ whose successful closure is $(\theta : \epsilon)$, then every computation of $\beta$ can reach such a success state (up to renaming-disjunction equivalence).

**Proof.** Every backtrack stack which is a descendent of $\beta'$ will contain $(\theta : \epsilon)$ as a closure, because no rules will eliminate it. By the Church-Rosser property, any computation can reach a descendent of $\beta'$.

This means that if one solution is found, this does not preclude the finding of other solutions. Thus we can characterise the behaviour of a goal stack by giving
the set of substitutions which can appear as solutions in successful computations (failing stacks taking the empty set).

2.2.3 Discussion

The major difference between SLD-resolution and SOS is that disjunctive information is made explicit. In SLD-resolution, each resolution step chooses one clause from the program; however, information about which clauses have been tried and which have yet to be tried is not explicit in the formal system. This means that information about how a particular system tries clauses must be represented outside the formal system (typically by a "search rule" in the search tree of candidate clauses [Llo84]). In SOS, clause information corresponds to the placement of the halves of a disjunction on the backtrack stack. As we will see, this facilitates the definition of sequential computations as variants of the single formal system.
The disadvantages of this "multi-step" style of operational semantics, in general, are that it is not very compositional, and that (except for the sequential variant) it is nondeterministic and therefore not a detailed description of an implementation.

I say that it is not very compositional because transitions depend not on the structure of high-level elements of the formal system (backtrack stacks) but on the structure of fairly low-level ones (individual formulae within goal stacks). Each transition involves a change to what may be only a small part of the elements of the system. This makes SOS a fairly clumsy system for proof purposes, since a lot of manipulation of structure is required in proofs involving it. Later, we will see other operational semantics which are more compositional; these will help to make clear some of the higher-level properties of SOS computations, and we will sometimes use these systems in the later soundness and completeness proofs.

The nondeterminism of SOS (which it shares with SLD-resolution) is a practical problem, because although the nondeterminism is used to model parallelism, it is not clear how this parallelism is to be implemented. However, there are so many ways of implementing parallelism (explicit breadth-first search or dovetailing on a rule-by-rule basis, process creation on a timesharing machine, actual parallel computation by parallel processors, etc.) that perhaps this is better left up to the implementor's choice.

2.3 Sequential Variants of SOS

SOS is an operational semantics in which nondeterminism is used to model parallelism. If we imagine an implementation in which each formula in each goal stack is assigned a processor, then a computation step corresponds to one of the processors completing the task of transforming the backtrack stack based on the form of its formula. The nondeterminism in the choice of formula reflects the fact that any of the processors may happen to finish its task first.
Thus, we can model a lower degree of parallelism in the implementation by restricting the choice of subgoal to be processed. Restricting the choosable subgoals to those in the first closure means that when a disjunction $B \lor C$ causes the backtrack stack to be split by an application of the $\lor$ rule, we must finish processing of $B$ before we can move on to $C$; the processing of the disjunction becomes sequential. Similarly, restricting the choosable subgoals to those on the top of their respective goal stacks means that in the processing of a conjunction $B \land C$, the computation of $B$ must finish before $C$ can be processed.

2.3.1 Definitions and Rules

**Definition 2.3.1** A computation $\beta \xrightarrow{\text{SQS}} \beta'$ is *sequential-or* if the subgoal selected for processing at each step is in the leftmost closure of the backtrack stack.

A computation sequence $\beta_0 \xrightarrow{\text{SQS}} \beta'$ is *sequential-and* if the subgoal selected for processing at each step is the leftmost subgoal in its goal stack ("at the top of its goal stack").

A computation sequence is *sequential-both* if it is both sequential-or and sequential-and.

We will call SOS restricted to sequential-or computations SOS/so, and SOS restricted to sequential-and computations SOS/sa. We will call SOS restricted to sequential-both computations SOS/sao, or more simply SP (Sequential Prolog).

There is only one sequential-both computation sequence of a given backtrack stack; that is, the operational semantics SP is *monogenic*. (We ignore renaming of variables here, assuming that our system has a deterministic algorithm for selecting variable names in the $\exists$ rule and for selecting an mgu in the unification algorithm.) In the case of sequential-both computations, the stacks really do behave as stacks, with the top of the stack to the left: nothing is relevant to the computation except the leftmost subgoal in the goal stack of the leftmost closure. Because only one choice of subgoal is possible, SP does not need to be implemented by more than one processor.
1. &:
   \((\theta : (B \& C), \alpha); \beta \xrightarrow{SP} (\theta : B, C, \alpha); \beta\)

2. v:
   \((\theta : (B \lor C), \alpha); \beta \xrightarrow{SP} (\theta : B, \alpha); (\theta : C, \alpha); \beta\)

3. ∃:
   \((\theta : (\exists x B), \alpha); \beta \xrightarrow{SP} (\theta : B[x := x'], \alpha); \beta\)
   where \(x'\) is some variable not appearing to the left of the arrow

4. Defined predicates:
   \((\theta : P(t_1, \ldots, t_n), \alpha); \beta \xrightarrow{SP} (\theta : A(t_1, \ldots, t_n), \alpha); \beta\)
   where \(\Pi\) contains the definition \((P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n))\)

5. Unification, success:
   \((\theta : s = t, \alpha); \beta \xrightarrow{SP} (\theta' : \alpha); \beta\)
   where \(\theta'\) is the mgu of \(s\theta\) and \(t\theta\)

6. Unification, failure:
   \((\theta : s = t, \alpha); \beta \xrightarrow{SP} \beta\)
   where \(s\theta\) and \(t\theta\) do not unify

**Figure 2–2:** The rules of the operational semantics SP.

Sequential-both computations are the most practically important class of computations, because many logic programming interpreters use only the sequential strategy. Because we will want to study sequential-both computations in great detail later, it will be useful to set down the rules of SP explicitly.

The rules of SP are as in Figure 2–2. The success states of SP are the backtrack stacks of the form \((\theta : \epsilon); \beta\). The single failure state of SP is the empty backtrack stack, \(\epsilon\).
2.3.2 Completeness Properties of Sequential Computations

Theorem 2.3.2 (SOS-Failure Completeness of SOS/so) If $\beta$ has a failing SOS-computation, then it has a failing sequential-or computation.

Proof. By induction on the length of the computation. Clearly a computation of zero steps is sequential-or. If the computation has one or more steps, then consider the first step in the failing computation which applies to the leftmost closure. If this is the first step of the computation, then by the induction hypothesis, the result holds.

Otherwise, the computation consists of an initial segment, the first step which applies to the leftmost closure, and a final segment. (There must be at least one step which applies to the leftmost closure, because the leftmost must fail.) We can form a new computation by taking $\beta$, performing the central step which replaces the leftmost closure $C$ by a stack $\beta'$; appending to that the initial segment, with $C$ replaced by $\beta'$ everywhere; and appending to that the final segment. The computation consisting of the altered initial segment and the final segment has, by the induction hypothesis, a failing sequential-or computation; so the original stack $\beta$ has a failing sequential-or computation.

Example. Consider the following computation:

\[
((0:0=1);(\exists x(2&x=3))) \Rightarrow ((0:0=1);(\exists x=2&x=3)) (3) \\
\Rightarrow ((0:0=1);(\exists x=2,x=3)) (&) \\
\Rightarrow (x=2,x=3) (\text{Unif, fail}) \\
\Rightarrow ([x:=3]:x=2) (\text{Unif, succ}) \\
\Rightarrow \epsilon (\text{Unif, fail})
\]

This computation can be transformed into the following sequential-or computation:

\[
((0:0=1);(\exists x(2&x=3))) \Rightarrow ((\exists x=2&x=3)) (\text{Unif, fail})
\]
So failing SOS-computations have SOS/so-analogues. Failing SOS-computations do not necessarily have SOS/sa-analogues, however. Consider the infinitely-looping predicate \( \text{Loop} \), defined by the predicate definition

\[
\text{Loop}() \leftrightarrow \text{Loop}()
\]

The query \( \text{Loop}() \& \text{false} \) fails in SOS but diverges for SOS/sa. (Recall that \textit{false} is an abbreviation for \( 0 = 1 \), where 0 and 1 are distinct closed terms.) However, there is an analogous relationship between sequential-and and sequential-both computations. This is not a straightforward consequence of the last theorem, because the set of failing computations of SOS/sa is a subset of that of SOS, and that of SP is a subset of that of SOS/so. However, the proof is very similar.

\textbf{Theorem 2.3.3 (SOS/sa-Failure Completeness of SP)} If \( \beta \) has a failing sequential-and computation, then it has a failing sequential-both computation.

\textbf{Proof.} As in the proof of the last theorem.

\( \square \)

The analogous property for success goes the other way: successful SOS-computations have sequential-and analogues, but not necessarily sequential-or analogues (the query \( \text{Loop}() \lor \text{true} \) is a counterexample).

First, a technical lemma.

\textbf{Lemma 2.3.4 (Independence of Closures for Success)}

(1) The backtrack stack \( \beta \) succeeds in SOS iff some element of \( \beta \) (that is, some closure in \( \beta \), taken as a singleton backtrack stack) succeeds in a smaller or equal number of steps.
(2) The backtrack stack $\beta$ succeeds in $\text{SOS/sa}$ iff some element of $\beta$ succeeds in $\text{SOS/sa}$ in a smaller or equal number of steps.

**Proof.** (1, $\rightarrow$) By induction on the length of the computation. If $\beta$ is a success state itself, then the result holds trivially. Otherwise, let $\beta \xrightarrow{\text{SOS}} \beta'$ be the first step. By the induction hypothesis, one of the closures in $\beta'$ succeeds. If this closure appears in $\beta$, then the result holds. Otherwise, it is the product of a computation step applied to a formula in a particular closure of $\beta$; therefore, that closure in $\beta$ also succeeds.

(1, $\leftarrow$) Say $\beta \equiv \beta_1; \theta \equiv C; \beta_2,$ and $C$ succeeds. Then we can get a successful computation of $\beta$ by taking the successful computation of $C$ and appending $\beta_1$ and $\beta_2$ on either side of each step.

(2) As in the proof for (1).

□

We mention the number of steps in this lemma, and at various points from hereon in, because of the requirements of later proofs by induction. They are not really essential for a high-level understanding of the theorems.

**Theorem 2.3.5 (SOS-Success Completeness of SOS/sa)** If $(\theta : \alpha)$ has a successful SOS-computation, then it has a successful sequential-and computation of smaller or equal length.

**Proof.** By induction on the length of the SOS-computation. If the length is 0, then the computation is clearly sequential-and. Otherwise, let the length be $n$. If the first step applies to a formula on the top of the goal stack, then the result follows immediately from the induction hypothesis.

Otherwise, the first step is not on a formula on the top of the goal stack. We have two subcases:

- The first step is not on a disjunction; that is, the computation begins $(\theta : A, \alpha) \xrightarrow{\text{SOS}} (\theta' : A, \alpha')$. By the induction hypothesis, the latter closure has
a sequential-and computation of length $n - 1$ or less. Again, we have two subcases.

- $A$ is not a disjunction. In this case, the sequential-and version of the tail of the computation begins $\langle \theta' : A, \alpha' \rangle^{\text{SOS}} \Rightarrow (\theta'' : \alpha'', \alpha')$. Then we can form a computation starting

$$\langle \theta : A, \alpha \rangle^{\text{SOS}} \Rightarrow (\theta'' : \alpha'', \alpha) \Rightarrow (\theta'' : \alpha'', \alpha')$$

and ending with the sequential-and computation from before. (Here, we use reasoning similar to that in the Church-Rosser property proof.) By a second application of the induction hypothesis, the tail of this computation (length $n - 1$) has a sequential-and computation; so the original closure has a sequential-and computation.

- $A \equiv B \lor C$. The sequential-and version of the tail of the computation begins $\langle \theta' : B \lor C, \alpha' \rangle^{\text{SOS}} \Rightarrow (\theta : B, \alpha') ; (\theta : C, \alpha')$. By Lemma 2.3.4, one of these closures succeeds with a sequential-and computation of length $n-2$ or less; and we can proceed as in the proof of the last subcase.

- The first step is on some disjunction $B \lor C$; that is, the computation starts

$$\langle \theta : \alpha_1, B \lor C, \alpha_2 \rangle^{\text{SOS}} \Rightarrow (\theta : \alpha_1, B, \alpha_2) ; (\theta : \alpha_1, C, \alpha_2)$$

One of these closures (by Lemma 2.3.4) succeeds after at most $n - 1$ steps. From this point, we can proceed as in the proof of the last subcase; that is, we can take the sequential-and version of the tail of this computation, interchange the first step of it with the disjunction step, and take the sequential-and version of this new tail.

\[ \square \]

**Corollary 2.3.6** If $\beta$ has a successful SOS-computation, then it has a successful sequential-and computation.

**Proof.** One closure in $\beta$ must have a successful SOS-computation, so by the theorem, that closure must have a successful sequential-and computation. We
can construct the sequential-and computation for $\beta$ by appending the extraneous closures to either side of each step.

The analogous relationship between sequential-or and sequential-both computations would be:

If $(\theta : a)$ has a successful sequential-or computation, then it has a successful sequential-both computation.

However, this result does not hold; consider the counterexample $(\text{Loop}() & false) \lor true$. This has a successful sequential-or computation, because the first disjunct fails due to parallel processing of $false$; but it has no successful sequential-both computation.

2.4 Compositional Operational Semantics

Here I address the problem of non-compositionality discussed earlier. There are operational semantics which have greater compositionality, and at least one with a high degree of compositionality. These operational semantics are in the form of "single-step" systems, in which the relation between initial state and final state ("solution state") is determined by a formal system, rather than being the transitive closure of a set of rewriting rules. (See Plotkin [Plo81] for more details about these styles of operational semantics.)

This style of operational semantics elucidates some of the higher-level properties of the basic SOS semantics, and will be used to some extent for proving, in the next chapters, the equivalences with the logical characterisations.

2.4.1 One-Stack Operational Semantics OS: Parallel Or

In the "one-stack" style of operational semantics OS, the relation $\Rightarrow$ is no longer between two backtrack stacks, but rather between a closure and either a substitu-
tion or the element fail. OS thus "abstracts out" some of the irrelevant details of SOS concerning backtrack stacks, and concentrates on the relationship between a closure and one of its solutions.

In the next section, we will see a very similar operational semantics, OSso, which describes a parallel-and, sequential-or system. OS describes a parallel-and, parallel-or system. The rules for OS are in Figure 2-3. The rules define a relation \( \mathcal{O}_S \) between a closure and either a substitution \( \rho \) or the expression fail. The production relation thus "jumps" from initial to final state in a single step; it is in the natural-deduction-style rules for the parts of the formulae that the computation takes place.

**Theorem 2.4.1** \( \beta \) has a successful computation in SOS iff some closure in it has a successful computation in OS.

**Proof.** (\( \rightarrow \)) By Lemma 2.3.4, some closure in \( \beta \) has a successful computation in SOS; we can do induction on the length of this SOS-computation. Each case follows fairly directly from the induction hypothesis. The only case that needs some care is the Unification, Failure case; there, we note that the failing closure cannot contribute to the success of \( \beta \), so it must be one of the other closures which succeeds.

(\( \leftarrow \)) We can build an SOS-computation from the OS-computation by reading out closures from bottom to top, sometimes adding the extraneous branches of disjunctions to the closure.

\( \Box \)

**Theorem 2.4.2** \( \beta \) has a failing computation in SOS iff all of its closures have failing computations in OS.

**Proof.** (\( \rightarrow \)) By induction on the length of the SOS-computation. Each case follows directly from the induction hypothesis.

(\( \leftarrow \)) By induction on the total number of steps in all the OS-computations; the cases are on the lowest step in the OS-computation of the first closure in \( \beta \).
Success rules.

\[=,1:\]
\[
(\theta : s = t) \overset{OS}{\Rightarrow} \theta' \\
(\theta : s = t) \overset{\theta \theta'}{\Rightarrow} s
\]
\[=,2:\]
\[
(\theta : s = t, \alpha_1, \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : s = t, \alpha_1, \alpha_2) \overset{\theta \theta'}{\Rightarrow} \rho
\]

\[\&:\]
\[
(\theta : \alpha_1, B, C, \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : \alpha_1, B \& C, \alpha_2) \overset{OS}{\Rightarrow} \rho
\]

\[\lor,1:\]
\[
(\theta : \alpha_1, B, \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : \alpha_1, B \lor C, \alpha_2) \overset{OS}{\Rightarrow} \rho
\]

\[\lor,2:\]
\[
(\theta : \alpha_1, C, \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : \alpha_1, B \lor C, \alpha_2) \overset{OS}{\Rightarrow} \rho
\]

\[\exists:\]
\[
(\theta : \alpha_1, B[x := x'], \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : \alpha_1, B, \alpha_2) \overset{OS}{\Rightarrow} \rho
\]

\[\exists:\]
\[
(\theta : \alpha_1, A(t_1, \ldots, t_n), \alpha_2) \overset{OS}{\Rightarrow} \rho \\
(\theta : \alpha_1, P(t_1, \ldots, t_n), \alpha_2) \overset{OS}{\Rightarrow} \rho
\]

Failure rules.

\[=,1:\]
\[
(\theta : s = t) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : s = t, \alpha_1, \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

\[=,2:\]
\[
(\theta : s = t, \alpha_1, \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

\[\&:\]
\[
(\theta : \alpha_1, B, \alpha_2) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : \alpha_1, B \& C, \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

\[\lor:\]
\[
(\theta : \alpha_1, B, \alpha_2) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : \alpha_1, C, \alpha_2) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : \alpha_1, B \lor C, \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

\[\exists:\]
\[
(\theta : \alpha_1, B[x := x'], \alpha_2) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : \alpha_1, B, \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

\[\exists:\]
\[
(\theta : \alpha_1, A(t_1, \ldots, t_n), \alpha_2) \overset{OS}{\Rightarrow} \text{fail} \\
(\theta : \alpha_1, P(t_1, \ldots, t_n), \alpha_2) \overset{OS}{\Rightarrow} \text{fail}
\]

Side-conditions:

\[(*)a\] \(\theta'\) is the mgu of \(s\theta\) and \(t\theta\)

\[(*)b\] \(x'\) is a variable not appearing in the conclusion

\[(*)c\] The definition \(P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n)\) appears in the program \(H\)

\[(*)d\] \(s\theta\) and \(t\theta\) do not unify

Figure 2-3: Rules for the operational semantics OS.
Each case follows directly from the induction hypothesis.

\(\Box\)

We can form a sequential-and variant of OS, called OS/sa, by restricting \(\alpha_1\) in all rules to be the empty goal stack. This variant has the same completeness properties with respect to SOS/sa as OS has to SOS; we can see this by noting that the SOS computations which the above theorems construct from OS/sa computations are sequential-and, and similarly in the other direction.

2.4.2 One-Stack Operational Semantics OSso: Sequential Or

There is also a sequential-or variant of the above operational semantics, called OSso. (I write its name this way to indicate that it actually has different rules; I retain the slash notation for restricted variants of other operational semantics, such as SOS/so.) It is identical to OS, except that the \(\vee, 2\) rule is replaced by the following rule:

\[
\text{\(\vee, 2:\)} \quad (\theta : \alpha_1, \mathcal{B}, \alpha_2) \xrightarrow{\text{OS}} \text{fail} \quad (\theta : \alpha_1, \mathcal{C}, \alpha_2) \xrightarrow{\text{OS}} \rho \\
(\theta : \alpha_1, \mathcal{B} \lor \mathcal{C}, \alpha_2) \xrightarrow{\text{OS}} \rho
\]

The same kind of relationship exists between OSso and SOS/so as between OS and SOS. The failure-completeness theorem is exactly the same as with OS and SOS:

**Theorem 2.4.3** \(\beta\) has a failing computation in SOS/so iff all of its closures have failing computations in OSso.

**Proof.** As above.

\(\Box\)

The success-completeness theorem has a different form, but is proven in essentially the same way. It depends on the failure-completeness theorem.

**Theorem 2.4.4** The backtrack stack \(C_1; C_2; \ldots; C_n\) has a successful computation in SOS/so iff there is some \(i, 1 \leq i \leq n\), such that \(C_j\) has a failing computation in OSso for all \(1 \leq j < i\), and \(C_i\) has a successful computation in OSso.
Proof. As above, except that in the case of a successful "or" computation we note that the SOS/so computation can and must fail on the left-hand branch before succeeding on the right-hand branch.

There is a sequential-and variant of OSso as well, OSso/sa, in which $\alpha_1$ must be empty for all rules. OSso/sa is sound and complete with respect to SP. Again, we can prove this by noting that the constructions that the above theorems make give SP computations from OSso/sa computations, and vice versa.

### 2.4.3 One-Formula Operational Semantics Csa

There is a very compositional equivalent of OS/sa, called Csa, in which the relation $\overset{\text{Csa}}{\Rightarrow}$ is not a binary relation between a closure and a substitution, but a ternary relation between an initial substitution, a single formula, and either a solution substitution or fail. Some of these rules have been presented before in a slightly different form [And89b] as the operational semantics called LP.

The rules for Csa are in Figure 2-4. The equivalence theorems between SOS/sa and Csa are as follows.

**Theorem 2.4.5** $(\theta : A_1, \ldots, A_n) \overset{\text{OS/sa}}{\Rightarrow} \rho$ iff there are substitutions $\theta_1, \ldots, \theta_{n-1}$ such that $(\theta : A_1 \overset{\text{Csa}}{\Rightarrow} \theta_1), (\theta_1 : A_2 \overset{\text{Csa}}{\Rightarrow} \theta_2), \ldots, (\theta_{n-2} : A_{n-1} \overset{\text{Csa}}{\Rightarrow} \theta_{n-1}), (\theta_{n-1} : A_n \overset{\text{Csa}}{\Rightarrow} \rho)$.

**Proof.** ($\rightarrow$) By induction on the size of the OS/sa-computation. Cases are on the bottommost rule application.

1, 2: The first step finds the most general unifier $\theta'$. The result follows immediately from the induction hypothesis.

&: Let $A_1 \equiv B \& C$. By the induction hypothesis, we have that $\theta : B \overset{\text{Csa}}{\Rightarrow} \theta'$ and $\theta' : C \overset{\text{Csa}}{\Rightarrow} \theta_1$; the result follows immediately.

V (1 and 2), $\exists$, P: the result follows directly from the induction hypothesis.
Success:

\[ \begin{align*}
\text{=} & : \theta : s = t \implies \theta \theta' \\
& \quad (\ast a) \quad \& : \theta : B \implies \theta' \quad \theta' : C \implies \rho \\
& \quad \theta : B \& C \implies \rho
\end{align*} \]

\[ \begin{align*}
\forall, \ 1 : & \quad \theta : B \implies \rho \\
& \quad \theta : B \lor C \implies \rho
\end{align*} \]

\[ \begin{align*}
\exists : & \quad \theta : B[x := x'] \implies \rho \\
& \quad \theta : \exists x \ B \implies \rho \\
& \quad (\ast b) \quad \forall : \quad \theta : A(t_1, \ldots, t_n) \implies \rho \\
& \quad \theta : P(t_1, \ldots, t_n) \implies \rho \\
& \quad (\ast c)
\end{align*} \]

Failure:

\[ \begin{align*}
\text{=} & : \theta : s = t \implies \text{fail} \\
& \quad (\ast d)
\end{align*} \]

\[ \begin{align*}
\&, \ 1 : & \quad \theta : B \implies \text{fail} \\
& \quad \theta : B \& C \implies \text{fail} \\
& \quad (\ast b) \quad \& , \ 2 : \quad \theta : B \implies \theta' \quad \theta' : C \implies \text{fail} \\
& \quad \theta : B \& C \implies \text{fail}
\end{align*} \]

\[ \begin{align*}
\exists : & \quad \theta : B \implies \text{fail} \\
& \quad \theta : C \implies \text{fail} \\
& \quad \exists : \quad \theta : B[x := x'] \implies \text{fail} \\
& \quad \theta : \exists x \ B \implies \text{fail} \\
& \quad (\ast b) \quad \forall : \quad \theta : A(t_1, \ldots, t_n) \implies \text{fail} \\
& \quad \theta : P(t_1, \ldots, t_n) \implies \text{fail} \\
& \quad (\ast c)
\end{align*} \]

Side-conditions:

(\ast a) \ \theta' \text{ is the mgu of } s\theta \text{ and } t\theta

(\ast b) \ x' \text{ is a variable not appearing in the conclusion}

(\ast c) \ \text{The definition } P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n) \text{ appears in the program } II

(\ast d) \ s\theta \text{ and } t\theta \text{ do not unify}

\[ \text{Figure 2–4: Rules for the Operational Semantics Csa.} \]
(→) By induction on the structure of the Csa-computations. Cases are on the form of $A_1$, and are straightforward.

The more significant corollary says that the set of successful queries is the same for the two operational semantics.

**Corollary 2.4.6** The query $A$ succeeds in SOS/sa iff it succeeds in Csa.

**Proof.** By the equivalence of OS/sa to SOS/sa, $(() : A)$ succeeds in SOS/sa iff it succeeds in OS/sa; and this happens, by the theorem, iff it succeeds in Csa.

We have the corresponding results for failure.

**Theorem 2.4.7** $(\theta_0 : A_1, A_2, \ldots, A_n)^{\text{SOS/sa}} \Rightarrow \text{fail}$ iff for some $i$, $1 \leq i \leq n$, we have that $(\theta_0 : A_1 \text{ Csa} \Rightarrow \theta_1), (\theta_1 : A_2 \text{ Csa} \Rightarrow \theta_2), \ldots, (\theta_{i-2} : A_{i-1} \text{ Csa} \Rightarrow \theta_{i-1}), (\theta_{i-1} : A_i \text{ Csa} \Rightarrow \text{fail})$.

**Proof.** (→) By induction on the bottommost rule application.

=, 1: by the induction hypothesis.

=, 2: immediate.

&: By the induction hypothesis, either $\theta_0 : B \text{ Csa} \Rightarrow \text{fail}$ (in which case $B \& C$ also fails); or $\theta_0 : B \text{ Csa} \Rightarrow \theta'$ and $\theta' : C \text{ Csa} \Rightarrow \text{fail}$ (in which case $B \& C$ also fails); or $\theta_0 : B \text{ Csa} \Rightarrow \theta'$ and $\theta' : C \text{ Csa} \Rightarrow \theta_2$ (in which case $\theta_0 : B \& C \text{ Csa} \Rightarrow \theta_1$), but one of the later $A_i$’s fails, and the result follows from the induction hypothesis.

\(\lor, \exists, \exists, P\): by the induction hypothesis.

(←) By induction on the structure of the Csa-computations. Cases are on the form of $A_1$, and are straightforward.

**Corollary 2.4.8** The query $A$ fails in SOS/sa iff it fails in Csa.
Proof. By the equivalence of OS/sa to SOS/sa, and by the theorem.

In summary, OS and its sequential-and variant OS/sa, OSso and its sequential-and variant OSso/sa, and Cs return some light on the structure of computations in SOS and its variants by making computations more compositional. However, SOS remains the only operational semantics we have found so far in which all four control disciplines are simple restricted variants of the main system.

SOS is also the simplest system, having only six rules; it seems that the various properties of success and failure brought out by the compositional operational semantics require more rules. Later, we will be proving the soundness and completeness of various proof-theoretic characterisations with respect to the operational semantics; to do this, we will sometimes use SOS because of the lower number of rules, and sometimes use one of the other operational semantics for their compositionality.

Finally, I should note that in all the equivalence theorems given in this section, whenever a computation in one system is constructed from several in another (or vice versa), the total number of rule applications is preserved in the other system. I did not include this information in the statements of the theorems or proofs because it would have cluttered them up unnecessarily; but readers should be able to see this by inspection of the proofs. We will need this property occasionally when we do inductions on the lengths of computations.

2.5 Some Properties of Existential Quantification

In the soundness and completeness proofs in later chapters, we will need some results about the behaviour of existential quantifiers. These, in turn, depend on more general results about the behaviour of backtrack stacks under substitutions. I group these results together in this section because they are essentially inductive proofs in the operational semantics.
2.5.1 Failure

Some properties of failure are shared by all variants of the operational semantics. One such property is that if a computation fails, it also fails under any more specific substitution.

Lemma 2.5.1 If $(\theta : \alpha)$ fails in SOS, then for all substitutions $\rho$, $(\theta \rho : \alpha)$ fails in SOS with a smaller or equal number of steps; and if the original computation was sequential-and (and/or sequential-or), so is the new computation.

Proof. By induction on the number of steps in the SOS computation, constructing a computation for $(\theta \rho : \alpha)$ from that for $(\theta : \alpha)$. Cases are on the form of the first rule.

&: Defined Predicate: Follows immediately from the induction hypothesis.

V: Both of the resultant closures must fail, with shorter computations than the original. Thus, by the induction hypothesis, the original closure fails.

$\exists$: We can choose $x'$ in the rule so that it does not appear in $\rho$ either.

Unification, success: One of the $\alpha$ formulae is $s = t$, and $s\theta$ and $t\theta$ unify. Let their unifier be $\theta'$. Now consider $s\theta\rho$ and $t\theta\rho$. Either they do not unify (in which case we have the result) or they have an mgu $\theta''$. But by the properties of mgu, since $\theta'$ is the mgu of $s\theta$ and $t\theta$, there must be a $\rho'$ such that $\theta'\rho' \equiv \rho\theta''$. Thus, by the induction hypothesis (taking $\rho$ to be $\rho'$), the result holds.

Unification, failure: One of the $\alpha$ formulae is $s = t$. Since $s\theta$ and $t\theta$ do not unify (there is no substitution that makes them identical), they cannot unify under any more specific substitution.

This lemma is required for an important theorem, which talks specifically about existential quantification.

Theorem 2.5.2 ("Fails-all-fail") If $(\theta : \exists x \ A)$ fails in SOS (or some variant thereof), then for all $t$, $(\theta : A[x := t])$ fails in SOS (or the variant) with a smaller number of steps.
Proof. The computation of \((\theta : \exists x A)\) must start by making one step to \((\theta : A[x := x'])\). By the lemma above, \((\theta[x' := t] : A[x := x'])\) must fail. But this closure behaves identically to \((\theta : A[x := t])\), in either SOS or SOS/sa.

So if the existential closure of a query fails, then every closed instance of it fails. But the converse is not true: it is not the case, in any variant of the operational semantics, that if all closed instances fail, then the existential closure fails. The paradigmatic counterexample here is the following. Consider the definition of the predicate "Inflist", which "succeeds" iff its argument is an infinite list:

\[
\text{Inflist}(x) \leftrightarrow \exists h \exists x t (x = [x h x t] \& \text{Inflist}(x t))
\]

This predicate clearly never does succeed, since we are still working in the domain of first order terms. However, whereas every closed instance of the query \(\text{Inflist}(y)\) fails, the query \(\exists y (\text{Inflist}(y))\) paradoxically diverges, looking for an infinite list but finding only ever-longer incomplete finite lists.

2.5.2 Success

A complementary set of results to the ones in the previous section are ones which deal with success: if a query succeeds, then it succeeds with all more specific substitutions that are still more general than the solution substitution.

Definition 2.5.3 We say that \(\theta'\) is more specific than \(\theta\), or that \(\theta\) is more general than \(\theta'\), if there is a \(\rho\) such that \(\theta' \equiv \theta \rho\).

Lemma 2.5.4 If \((\theta : \alpha)\) succeeds in SOS with the solution \(\theta_f\), then, for any \(\theta'\) more specific than \(\theta\) but more general than \(\theta_f\), \((\theta' : \alpha)\) succeeds in SOS with the solution \(\theta_f\) with a smaller or equal number of steps. Moreover, if the original computation was sequential-and (and/or sequential-or), so is the new computation.

Proof. By induction on the number of steps in the SOS-computation. Cases are on the first step.
&, Defined Predicates: Follows directly from the induction hypothesis.

\( V \): If the original computation was parallel-or, then one of the resultant closures must succeed (by Lemma 2.3.4), so the result follows by the induction hypothesis. If the original computation was sequential-or, then either the first closure succeeds (in which case the result follows by the induction hypothesis), or else the first closure fails (in which case, by the fails-all-fail lemma, it fails under any more specific substitution) and the second succeeds (so the result follows from the induction hypothesis).

\( \exists \): We can choose \( x' \) so that does not appear in \( \theta' \) either; then the result follows from the induction hypothesis.

Unification, success: Let \( \theta' \equiv \theta \rho \). One of the \( \alpha \) formulae is of the form \( s = t \), and \( s \theta \) and \( t \theta \) have an mgu, \( \rho' \). We can prove by a simple induction that \( s \theta \rho = t \theta \rho \), and \( \theta \rho \) is more specific than \( \theta' \); so \( s \) and \( t \) unify under \( \theta' \) too; let their mgu be \( \theta'' \). Thus, \( s \theta \rho \theta'' = t \theta \rho \theta'' \); by the properties of mgu (\( \rho' \)), there must be a \( \rho'' \) such that \( \rho' \rho'' \equiv \rho \theta'' \). By the induction hypothesis (on \( \rho'' \)), we have the result.

Note that there is no case for failure of unification, as we have assumed that the closure succeeds.
\( \Box \)

**Theorem 2.5.5** ("Succeeds-one-succeeds") If \((\theta : \exists x \ B)\) succeeds in SOS (or some variant thereof) with solution \( \theta_f \), then there is a closed \( t \) such that \((\theta : B[x := t])\) succeeds in SOS (or the variant) with a smaller number of steps.

**Proof.** Let the first step in the computation replace \( x \) by \( x' \), and let \( t \) be any closed instance of \( x' \theta_f \). By a simple induction we can prove that \((\theta : \exists x (B & x = t))\) succeeds with solution \( \theta_f [x' := t] \). But then by the lemma, \((\theta[x' := t] : \exists x (B & x = t))\) succeeds, after making two steps to \((\theta[x' := t] : B \land x = x', x' := t)\); this closure will behave essentially identically to \((\theta : B[x := t])\), which must therefore succeed.
\( \Box \)
Thus, if the existential closure of a query succeeds, then there is some instance of it which succeeds.

2.5.3 Solution Completeness

The converse of the theorems in the last section is one which says that if a solution exists, the system will find one. This is true only in the case of parallel-or systems. It is a bit more convenient to prove this using OS rather than SOS.

Lemma 2.5.6 If $\theta'$ is more specific than $\theta$, and $(\theta' : \alpha) \osf{=}_{\theta} \theta_j$, then there is a $\theta'_j$ such that $(\theta : \alpha) \osf{=}_{\theta} \theta'_j$. Moreover, if the original computation was sequential-and, then so is the new computation.

Proof. By induction on the number of rule applications in the OS-computation. Cases are on the bottommost rule applied.

Success, $\neg$, 1, Success, $\neg$, 2: One of the $\alpha$ formulae is $s = t$, and the mgu of $s\theta'$ and $t\theta'$ is some $\rho$. Thus we know that $s\theta' \rho \equiv t\theta' \rho$, so $s\theta$ and $t\theta$ unify, resulting in a substitution which is at least as specific as $\theta' \rho$. In the case of $\neg$, 1, the result follows immediately; in the case of $\neg$, 2, the result follows from the induction hypothesis.

Success, $\land$; $\lor$, 1; $\lor$, 2; $\exists$; $P$: follows directly from the induction hypothesis. In the $\exists$ case, if the new variable does not appear in $\theta'$, then it does not appear in $\theta$ either. 

$\square$

Theorem 2.5.7 ("One-succeeds-succeeds") If there is a $t$ such that $(\theta : B[x := t])$ succeeds in SOS (resp. SOS/sa), then $(\theta : \exists x B)$ succeeds in SOS (resp. SOS/sa).

Proof. The first step of the computation of $(\theta : \exists x B)$ is to $(\theta : B[x := x'])$. But we know that $(\theta[x' := t] : B[x := x'])$ succeeds; so by the lemma, and the
equivalence of OS (OS/sa) to SOS (SOS/sa), \((\theta : B[x := x'])\) succeeds as well.

This theorem does not work in the sequential-or case, because there the rule \(V, 2\) depends upon a failing computation of the left-hand disjunct, which may not exist. For instance, although the closure \((() : Inplist(3) \lor 3 = 3)\) succeeds in both OS and OSso, the closure \((() : \exists x (Inplist(x) \lor x = x))\) succeeds only in OS, and diverges in OSso.

Finally, a note about the statements of all the theorems in this section: wherever they mention one computation taking fewer steps than another, the result applies not only to the SOS variants mentioned, but also to any equivalent compositional operational semantics. This is because the number of rule applications in the compositional semantics computations is the same as the number of steps in the corresponding SOS variant.

2.6 Summary and Classification of Queries

I have given a number of different operational semantics here, but they fall into four basic categories: namely, the four different combinations of parallel or sequential "and" and parallel or sequential "or". The OS variants and Csa fall into one category or another depending on their equivalence properties with respect to the SOS variants. Figure 2–5 summarises this classification.

Queries themselves can be classified by how they behave in the various control disciplines. Figure 2–6 summarises this information. There are two classes of failing queries: one class all of whose queries fail in all control disciplines, and a wider one whose queries fail in all parallel-and control disciplines. There are three classes of successful queries: one all of whose queries succeed in all disciplines, a wider one all of whose queries succeed in all disciplines except SP, and a still wider one all of whose queries succeed in all parallel-or disciplines.
Chapter 2. Operational Semantics

<table>
<thead>
<tr>
<th>Par. “Or”</th>
<th>Par. “And”</th>
<th>Stack-of-Stacks</th>
<th>One-Stack</th>
<th>One-Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOS</td>
<td>OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq. “And”</td>
<td>SOS/sa</td>
<td>OS/sa</td>
<td>Csa</td>
<td></td>
</tr>
<tr>
<td>Seq. “Or”</td>
<td>Par. “And”</td>
<td>SOS/so</td>
<td>OS/so</td>
<td></td>
</tr>
<tr>
<td>Seq. “And”</td>
<td>SOS/soa (“SP”)</td>
<td>OS/so/sa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2–5: Classification of operational semantics.

<table>
<thead>
<tr>
<th>Failure</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 2–6: Classification of queries as to their behaviour in the variants of SOS. A, B, and C are example formulae within these classes; A \( \equiv \text{Loop() \& false} \); B \( \equiv (\text{Loop() \& false}) \lor \text{true} \); C \( \equiv \text{Loop()} \lor \text{true} \).
Chapter 3

Characterising Parallel Systems

In this chapter, I present a sequent calculus which characterises the outer two circles of the Venn diagram in Figure 2-6. In this calculus, we can state and prove assertions about the success and failure of queries in parallel logic programming systems. Assertions can be \textit{signed formulae}, which are essentially expressions of the form $S(A)$ ("A succeeds") or $F(A)$ ("A fails"); assertions can also be expressions built up from signed formulae with the usual connectives of first order logic.

The sequent calculus is in two distinct parts. LKE is a very typical classical sequent calculus with equality, and deals with connectives at the top level of assertions. PAR is a set of simple axioms which describe how the goal-formula connectives distribute over the signs $S$ and $F$; for instance, $S(B&C) \rightarrow S(B) \& S(C)$. LKE will also be used in the next chapter, where we shall deal with sequential systems; but there, PAR will be replaced by a set of analogous axioms, SEQ.

Associated with assertions and sequents is a notion of validity. For reasons which will be discussed in Chapter 5, we cannot have a finitary sequent calculus which is both sound (all derivable sequents are valid) and complete (all valid sequents are derivable); however, the calculus in this chapter is sound, and has various useful completeness properties weaker than full completeness.

This sequent calculus can therefore be seen as a semantic characterisation of parallel Prolog; or, conversely, as a logic with respect to which parallel Prolog enjoys soundness and completeness properties. It can also be used to prove prop-
properties of programs. These issues will be discussed in greater detail in the final section of this chapter.

3.1 Overview and Definitions

I should first give a detailed overview of what we are trying to accomplish and how.

Our main concern is that of characterising, via proof systems, the success or failure of queries to the parallel logic programming system SOS. We could do this by trying to characterise the sequence of satisfying substitutions from a goal formula, but this inevitably seems to lead us back to the realm of operational semantics. If we instead stick to just characterising whether the query succeeds or fails, we can express the behaviour of a goal formula $A$ by expressions of the form $S(A)$ or $F(A)$, to be read as "$A$ succeeds" or "$A$ fails", respectively. These expressions will be called signed (goal) formulae. (For technical reasons, as we will see, it will be necessary to distinguish the subclass of queries which fail without making predicate calls.)

Signed formulae could act as the judgments of our proof system, but we often will want to make more complex assertions about success or failure: "For all $x$ there is a $y$ such that $P(x,y)$ succeeds," or "For all $l$, if $List(l)$ succeeds then $\exists n \text{ Length}(l,n)$ succeeds," for instance. We can therefore define assertions as a class of pseudo-formulae built up from signed formulae and equality formulae by the classical connectives. In assertions, as we will see, the $S$ and $F$ signs act much like modalities; but we cannot define the syntax of assertions as we would define formulae in modal logic, because only goal formulae can appear within a sign.

Finally, we need to decide what style of proof system to use: natural deduction, sequent calculus, tableaux, or some other style. I have chosen to use the sequent calculus style because it seems to provide more uniformity and expressive power than the natural deduction style, and because its organisation of rules is more natural for our purposes than the tableau style.
We therefore have the following definitions.

Definition 3.1.1 A **signed formula** is an expression of one of the forms \( S(A) \), \( F^Y(A) \), or \( F^N(A) \), where \( A \) is a goal formula. The informal meaning of these expressions is intended to be, respectively, "\( A \) succeeds", "\( A \) fails, possibly performing predicate calls", and "\( A \) fails without performing predicate calls". We will sometimes use the notation \( F(A) \) to mean either \( F^Y(A) \) or \( F^N(A) \), when either it does not matter which we mean or it is temporarily being used to mean consistently one thing. Similarly, we use the notation \( \sigma(A) \) to mean either \( S(A) \), \( F^Y(A) \), or \( F^N(A) \).

An **assertion** is an expression of the following BNF syntax:

\[
A ::= A_1 \& A_2 \mid \neg A \mid \exists x A \mid s = t \mid \sigma(G)
\]

where \( G \) is a goal formula. We will generally use \( A, B, C, D \) as metavariables ranging over assertions as well as goal formulae; their use will be unambiguous.

We will define \( B \lor C \) as \( \neg(\neg B \& \neg C) \), \( B \supset C \) as \( \neg(B \& \neg C) \), and \( \forall x B \) as \( \neg(\exists x \neg B) \).

Notions of **free and bound variables** for signed formulae and assertions will be a straightforward extension of those for formulae.

A **sequent** is an expression of the form

\[
A_1, \ldots, A_n \rightarrow D_1, \ldots, D_m
\]

where \( n, m \geq 0 \) and each of the \( A_i \)'s and \( D_j \)'s is an assertion. We will treat each side of the sequent as a finite set of assertions. \( \Gamma \) and \( \Delta \) will range over sequences/sets of assertions: we will write \( \Gamma, \Delta \) for the union of the two sets \( \Gamma \) and \( \Delta \); \( \Gamma, A \) for the union of \( \Gamma \) with \( \{A\} \); and so on.

We will generally refer to the sequence of formulae on the left-hand side of the arrow as the **antecedent**, and that on the right-hand side as the **consequent** of the sequent.
We now need to have a notion of validity which expresses our intended interpretation of the truth of assertions. In keeping with the approach of basing the logic on the operational semantics, I will give an inductive definition of validity of a closed assertion, at the base of which are simple notions of validity of closed equality formulae and closed signed goal formulae.

In this chapter, we will be concerned with validity with respect to the parallel operational semantics SOS. In the next chapter, the sequential operational semantics SP will concern us, but the definition of validity will be essentially the same, so I will express this notion in more general terms.

Definition 3.1.2 A closed assertion is valid with respect to a particular operational semantics $O$ (chosen from the variants of SOS) and program $H$ just in the following cases.

- $B \& C$ is valid iff both $B$ and $C$ are valid.
- $\neg B$ is valid iff $B$ is not valid.
- $\exists x\ B$ is valid iff there is a closed term $t$ such that $B[x := t]$ is valid.
- $s = t$ is valid iff $s$ and $t$ are identical.
- $S(A)$ is valid iff the backtrack stack $(()) : A$ succeeds in the operational semantics $O$.
- $F^Y(A)$ is valid iff the backtrack stack $(()) : A$ fails in the operational semantics $O$.
- $F^N(A)$ is valid iff the backtrack stack $(()) : A$ can fail in the operational semantics $O$ without performing any Defined Predicate steps.

Note that the negation sign has nothing to do with negation as failure in this context; $\neg S(A)$ means that $A$ does not succeed, which might mean either that it fails or that it diverges.
We will also speak of the validity of a sequent, which is based on the validity of formulae and the notion of substitution.

Definition 3.1.3 A sequent $A_1, \ldots, A_n \rightarrow D_1, \ldots, D_m$ is valid iff for every $\theta$ which makes all the assertions in the sequent closed, either some $A_i \theta$ is not valid or else some $D_j \theta$ is valid.

It should be clear that the sequent $A_1, \ldots, A_n \rightarrow D_1, \ldots, D_m$ is valid iff the formula $\forall [A_1 \& \ldots \& A_n \supset D_1 \lor \ldots \lor D_m]$ is valid, as in the usual presentations of the classical sequent calculus.

The notion of failure without performing Defined Predicate steps will recur frequently. For brevity, in this context I will use the expression “flat failure”, and say that a query “fails flatly”.

The effect of these definitions is to say that the sequent calculi which we will build will be program logics: logical systems in which properties of programs and queries can be stated and proved. Like Floyd-Hoare and other program logics [Gol82], the syntactic elements of programs and queries will appear in the judgments of the proof systems, as sub-expressions of these judgments. Unlike traditional program logics for imperative and functional programming, however, the programming language being talked about will itself be logical.

In fact, since we have separate definitions for goal formulae and assertions, we could have chosen to use syntactically distinct connectives within goals (say $\&$, $\lor$, $\exists$) to emphasise this separation of levels. The task of the program logic will be to clarify the correspondence between the connectives in queries and programs (the connectives within the signs) and those in assertions (the connectives outside the signs).
3.2 LKE and Its Soundness

The sequent calculus in this chapter will have two parts: a set of rules concerned only with assertions, which will also be used in the next chapter, and a set of axioms concerning signed formulae, which will be used only in this one. The first part is the subject of this section.

The assertion rules are collectively called LKE, because they are essentially Gentzen's LK ("Logischer Klassischer," or classical-logic) sequent calculus [Gen69], augmented with Equality rules. Readers will recognise the similarity between these rules and those found in Gentzen's original description, given the common modern practice of considering each side of a sequent as a set.

The rules of LKE are in Figures 3–1 and 3–2, classified into groups. We will generally refer to the upper sequents in any application of a rule as the premisses of the rule, and the lower sequent as the conclusion. We will find it very useful to have rules for the defined connectives $\lor$, $\land$, and $\forall$, which are in Figure 3–3. These rules are easily derivable by expanding the connectives in question into their definitions.
1. Equality, first group.

Eq: \[ \frac{\Gamma \rightarrow t = t, \Delta}{\Gamma \rightarrow t = t, \Delta} \]

Ineq: \[ \frac{\Gamma, f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m) \rightarrow \Delta}{\Gamma, f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m) \rightarrow \Delta} \quad (*)a \]

Comp: \[ \frac{\Gamma, s_1 = t_1, \ldots, s_n = t_n \rightarrow \Delta}{\Gamma, f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \rightarrow \Delta} \]

2. Connectives.

\&, l: \[ \frac{\Gamma, B, C \rightarrow \Delta}{\Gamma, B \& C \rightarrow \Delta} \]
\&, r: \[ \frac{\Gamma \rightarrow B, \Delta}{\Gamma \rightarrow B \& C, \Delta} \]

\exists, l: \[ \frac{\Gamma, B[x := y] \rightarrow \Delta}{\Gamma, \exists x \ B \rightarrow \Delta} \quad (*)b \]
\exists, r: \[ \frac{\Gamma \rightarrow B[x := t], \Delta}{\Gamma \rightarrow \exists x \ B, \Delta} \]

\neg, l: \[ \frac{\Gamma \rightarrow B, \Delta}{\Gamma, \neg B \rightarrow \Delta} \]
\neg, r: \[ \frac{\Gamma, B \rightarrow \Delta}{\Gamma \rightarrow \neg B, \Delta} \]

Side-conditions:

(*a) \( f \not\equiv g \)

(*b) \( y \) does not occur free in the lower sequent

Figure 3-1: Rules for proof system LKE, part I.
3. Equality, second group.

Occ: \[
\frac{\Gamma, s = t \rightarrow \Delta}{\Gamma, s = t, A(s) \rightarrow \Delta}
\]  (*c)

Sub, l: \[
\frac{\Gamma, s = t, A(s) \rightarrow \Delta}{\Gamma, s = t, A(t) \rightarrow \Delta}
\]  Sub, r: \[
\frac{\Gamma, s = t, A(s) \rightarrow \Delta}{\Gamma, s = t, A(t) \rightarrow \Delta}
\]


Ax: \[
\frac{\Gamma, A \rightarrow A, \Delta}{\Gamma, A \rightarrow A, \Delta}
\]

Cut: \[
\frac{\Gamma_1 \rightarrow A_1 \Delta_1 \quad \Gamma_2, A \rightarrow \Delta_2}{\Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2}
\]

Thin, l: \[
\frac{\Gamma \rightarrow \Delta}{\Gamma, A \rightarrow \Delta}
\]  Thin, r: \[
\frac{\Gamma \rightarrow \Delta}{\Gamma \rightarrow A, \Delta}
\]

Side-conditions:

(*c) s is a proper subterm of t or vice versa

Figure 3-2: Rules for proof system LKE, part II.

5. Derivable rules for defined connectives.

\(\forall\), l: \[
\frac{\Gamma, B \rightarrow \Delta \quad \Gamma, C \rightarrow \Delta}{\Gamma, B \lor C \rightarrow \Delta}
\]  \(\forall\), r: \[
\frac{\Gamma \rightarrow B, C, \Delta}{\Gamma \rightarrow B \lor C, \Delta}
\]

\(\exists\), l: \[
\frac{\Gamma \rightarrow B, \Delta \quad \Gamma, C \rightarrow \Delta}{\Gamma, B \lor C \rightarrow \Delta}
\]  \(\exists\), r: \[
\frac{\Gamma, B \rightarrow C, \Delta}{\Gamma \rightarrow B \lor C, \Delta}
\]

\(\forall\), l: \[
\frac{\Gamma, B[x := t] \rightarrow \Delta}{\Gamma, \forall x B \rightarrow \Delta}
\]  \(\forall\), r: \[
\frac{\Gamma \rightarrow B[x := y], \Delta}{\Gamma \rightarrow \forall x B, \Delta}
\]  (*b)

Side-conditions:

(*b) y does not occur free in the lower sequent

Figure 3-3: Derivable rules for defined connectives in LKE.
We would like to prove the soundness of LKE alone, but actually we will want to prove something somewhat stronger than that: that whenever we add new rules to the system (as we will do twice later), the LKE rules remain sound, and that if we change the operational semantics at the base of the definition of validity, the LKE rules remain sound. The following theorem proves this.

Theorem 3.2.1 (Soundness of LKE Rules) If all the premisses of an application of an LKE rule are valid with respect to an operational semantics $\mathcal{O}$ and program $\Pi$, then so is the conclusion.

Proof. We can analyse each case separately. I will give only representative cases here.

Eq: If an assertion $t = t$ appears in the consequent of a sequent, then the sequent is valid, because under any substitution there will be a valid assertion in the consequent, regardless of whether there is a valid assertion in the antecedent.

&, l: Assume that the premiss is valid, and that under a given substitution $\theta$ that makes all assertions closed, all the antecedent assertions of the conclusion are valid. Then it must be the case that $B\theta$ and $C\theta$ are both valid, so all the antecedent assertions of the premiss are valid under $\theta$; but since the premiss is valid, one of the consequent assertions of the premiss (which is the same as one of the consequent assertions of the conclusion) must be valid under $\theta$ as well.

&, r: Assume that the premisses are valid. Under a given substitution $\theta$ which makes all the assertions closed, if all the antecedent assertions in either premiss are valid, then so is one of its consequent assertions. If this consequent assertion is one of the $\Delta$ assertions, then one of the consequent assertions of the conclusion is also valid under $\theta$. If not, then both $B\theta$ and $C\theta$ must be valid, so $(B&C)\theta$ must be valid, and again one of the consequent assertions of the conclusion is valid.

$, l: Assume that the premiss is valid. Let $\theta$ be a substitution which makes the conclusion closed; then all substitutions which make the premiss closed will be of the form $\theta[y := t]$. Given $\theta[y := t]$, there are two possibilities: either one of the assertions in the consequent of the premiss is valid, or one of its antecedent
assertions is not valid. In the first case, one of the consequent assertions in the conclusion is valid under \( \theta \) (since none of the conclusion assertions contain \( y \)). In the second case, if the invalid assertion is in the \( \Gamma \) assertions, then there is an invalid assertion in the antecedent of the conclusion as well.

Otherwise, none of the consequent assertions (none of which contain \( y \)) are valid under \( \theta[y := t] \), all the antecedent assertions which do not contain \( y \) are valid under \( \theta[y := t] \), and \((B[x := y])\theta[y := t]\) is invalid. But then for the premiss to be valid, \((B[x := y])\theta[y := t]\) must be invalid for any choice of \( t \). This means that for any choice of \( t \), \( B[x := t] \theta \) is invalid; \((B)\theta\) must therefore be invalid, and again there is an invalid assertion in the antecedent of the conclusion.

Sub, \( r \): Under any substitution \( \theta \) which makes the sequents closed, either \( s \) and \( t \) are not identical (thus making \( s = t \) invalid and the conclusion valid), or else \( s \) and \( t \) are identical and the two sequents are also identical.

Cut: Assume that the premisses are valid, and that \( \theta \) makes the sequents closed. If one of the \( \Gamma_1 \) or \( \Gamma_2 \) assertions were invalid under \( \theta \), or one of the \( \Delta_1 \) assertions were valid under \( \theta \), the conclusion would be valid under \( \theta \). Otherwise, \( \Delta \theta \) must be valid for the first premiss to be valid; but then one of the \( \Delta_2 \) assertions must be valid for the second premiss to be valid, and again the conclusion is valid under \( \theta \).

\( \therefore, r \): We have the following derivation for this rule. (We treat \( B \supset C \) as \( \neg(\neg B \land C) \)).

\[
\frac{
\Gamma, B \rightarrow C, \Delta
}{\Gamma, \neg C \rightarrow \Delta}
\]

\[
\frac{
\Gamma, B \land \neg C \rightarrow \Delta
}{\Gamma \rightarrow \neg(\neg B \land C), \Delta}
\]

Other cases are similar.

\[ \Box \]

I will leave the detailed historical and philosophical discussion of these rules to the end of the chapter, and go on with the technical material.
3.3 Axioms for Parallel Validity

In this section, I will present and prove valid a set of axioms describing the behaviour, in parallel systems, of the connectives within the $S$ and $F$ signs. These axioms, collectively called PAR, will take the form of sequents. They are intended to be used with the "cut" rule from LKE in the derivation of the success and failure of goal formulae, and in later sections I will prove that this set of laws is complete for this purpose.

The axioms can be seen as distributive laws, because they define how the signs "distribute" over the goal formula connectives. Indeed, the laws for the $F$ signs resemble De Morgan's laws of the distribution of negation over the first order connectives.

3.3.1 Predicate Unfoldings and the $F^N$ sign

First, some more elucidation of the purpose of the $F^N$ sign is necessary; this will motivate the definition of predicate unfoldings of formulae, which will play an important role in the parallel axioms.

We want to give a "logical" definition of the behaviour of the existential quantifier under failure. In traditional sequent calculi, and in LKE itself, this corresponds to the behaviour of $\exists$ on the left-hand side of a sequent. We would therefore like to be able to assert that $\forall x \ F(B) \rightarrow F(\exists x \ B)$, so that we could (with Cut) replace $F(\exists x \ B)$ on the right-hand side by $\forall x \ F(B)$.

However, because of the Inflist paradox (page 47), this sequent is not, in fact, valid; there are goal formulae $B$ such that every instance of $B$ fails (and thus $\forall x \ F(B)$ is valid) but $\exists x \ B$ diverges.

There are various approaches to a solution to this problem, some of which will be discussed at the end of this chapter. The one followed here is to make a distinction between failure which may involve predicate calls, and failure without
predicate calls. The \textit{Inflist} paradox arises because there is no bound to the number of predicate calls that a given instance of \( F(\text{Inflist}(x)) \) can make before failing. The computation of the existential closure essentially tries all possibilities, and thus keeps looking forever. If, however, every instance of a query fails making fewer than a fixed number of predicate calls, then there are only "a finite number of ways" in which instances can fail, so the computation of the existential closure will terminate after trying them all.

We therefore have that \( \forall x \; F^N(B) \to F(\exists x \; B) \) is a valid sequent; if all instances fail without making any predicate calls at all, then the existential closure fails. This rule is not useful for \( B \)'s in which the computation \textit{does} involve predicate calls, unless there is some way of converting a query which makes predicate calls into one which does not. Fortunately, because of the way in which we have defined predicate definition, this conversion process can be done easily by expanding predicate calls appearing in the query into their predicate bodies. This is just the familiar "unfolding" operation from Burstall and Darlington [BD77] or Tamaki and Sato [TS84]; later, we will meet a different kind of unfolding which is necessary for characterising sequential systems.

\textbf{Definition 3.3.1} An assertion (formula) \( A' \) is a \textit{predicate 1-unfolding} of another assertion (formula) \( A \) if it is \( A \) with one occurrence of a subformula \( P(t_1, \ldots, t_n) \) replaced by an occurrence of \( A(t_1, \ldots, t_n) \), where \( P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n) \) is in the program \( \Pi \).

\( A' \) is a \textit{predicate unfolding} of \( A \) if it is \( A \) with zero or more successive predicate 1-unfoldings applied to it.

In the rules which follow, we will allow the replacement of a formula or assertion by its predicate unfolding and vice versa. The following theorem will help to justify this.

\textbf{Theorem 3.3.2 (Operational Equivalence of Predicate Unfoldings)} If \( A' \) is a predicate unfolding of \( A \), then any backtrack stack \( \beta \) containing \( A \) succeeds (fails) in any of the SOS variants iff \( \beta \) with \( A \) replaced by \( A' \) succeeds (fails).
Proof. By induction on the number of predicate 1-unfoldings from A to A', and on the length of the computation of β. The only non-trivial case is when A is itself a predicate call, and is the subject of the first step. In this case, it should be clear that a Defined Predicate expansion step can be inserted or deleted to form one computation from the other.

3.3.2 The PAR Axioms and their Validity

The laws are in Figure 3-4, presented as a set of axioms, which are in turn just a special kind of rule. We will refer to this set of rules as PAR, since they have to do with the parallel connectives; they are intended to be used in conjunction with the rules from LKE, and we will refer to the combined system as LKE+PAR.

The PAR axioms are in Figure 3-4, classified by the sign involved, the side of the sequent on which we will generally use them, and the connective immediately within the sign. For examples of the use of LKE+PAR to derive sequents, see Appendix A.

As with the LKE rules, we must prove these rules sound – that is, that sequents derivable from valid sequents are also valid. (For brevity, “validity” in this chapter will consistently mean “validity with respect to SOS”.)

Theorem 3.3.3 (Validity of PAR axioms) Each instance of the axiom schemata in PAR is a valid sequent with respect to the operational semantics SOS.

Proof. One case for each arrow direction of each axiom.

S(=): For a given substitution θ which makes the sequent closed, S(s = t) is valid under θ if and only if sθ and tθ unify; but since sθ and tθ are both closed, this happens iff they are identical, that is, iff s = t is valid under θ as well.

For the rest of the cases, we will assume that θ is a substitution which makes the sequent closed.
### Chapter 3. Characterising Parallel Systems

#### Success axioms:

<table>
<thead>
<tr>
<th>$S(...)</th>
<th>\text{left}</th>
<th>\text{right}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: \quad S(s=t) \rightarrow s = t</td>
<td>s = t \rightarrow S(s=t)</td>
<td></td>
</tr>
<tr>
<td>$&amp;: \quad S(B &amp; C) \rightarrow S(B) &amp; S(C)</td>
<td>S(B) &amp; S(C) \rightarrow S(B &amp; C)</td>
<td></td>
</tr>
<tr>
<td>$\lor: \quad S(B \lor C) \rightarrow S(B) \lor S(C)</td>
<td>S(B) \lor S(C) \rightarrow S(B \lor C)</td>
<td></td>
</tr>
<tr>
<td>$\exists: \quad S(\exists x \ B) \rightarrow \exists x \ S(B)</td>
<td>\exists x \ S(B) \rightarrow S(\exists x \ B)</td>
<td></td>
</tr>
</tbody>
</table>

#### Failure axioms:

<table>
<thead>
<tr>
<th>$F(...)</th>
<th>\text{left}</th>
<th>\text{right}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: \quad F(s = t) \rightarrow \neg s = t</td>
<td>\neg s = t \rightarrow F(s = t)</td>
<td></td>
</tr>
<tr>
<td>$&amp;: \quad F(B &amp; C) \rightarrow F(B) \lor F(C)</td>
<td>F(B) \lor F(C) \rightarrow F(B &amp; C)</td>
<td></td>
</tr>
<tr>
<td>$\lor: \quad F(B \lor C) \rightarrow F(B) &amp; F(C)</td>
<td>F(B) &amp; F(C) \rightarrow F(B \lor C)</td>
<td></td>
</tr>
<tr>
<td>$\exists: \quad F(\exists x \ B) \rightarrow \forall x F(B) (*a)</td>
<td>\forall x F(N)(B) \rightarrow F(\exists x \ B)</td>
<td></td>
</tr>
</tbody>
</table>

#### Miscellaneous axioms:

<table>
<thead>
<tr>
<th>$F^N / F^Y:</th>
<th>\text{left}</th>
<th>\text{right}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma(A) \rightarrow \sigma(A')</td>
<td>\sigma(A') \rightarrow \sigma(A) (*b)</td>
<td></td>
</tr>
</tbody>
</table>

#### Notes:

(*a): These rules are asymmetric; all of the other Success and Failure axiom pairs are symmetric.

(*b): Side-condition: $A'$ is a predicate unfolding of $A$, and $\sigma$ is either $S$ or $F^Y$.

**Figure 3-4:** PAR axioms characterising parallel connectives. $F$ means either $F^Y$ or $F^N$, its use being consistent throughout each axiom.
Chapter 3. Characterising Parallel Systems

$S(\&)$, left: If $B \& C$ succeeds under $\theta$, this must mean that the backtrack stack $((() : B\theta, C\theta))$ succeeds. But since both are closed under $\theta$, this can happen only if both succeed independently; that is, if both $S(B)\theta$ and $S(C)\theta$ (and thus $S(B \& C)\theta$) are valid.

$S(\&)$, right: If $S(B) \& S(C)$ is valid, then both $B$ and $C$ must succeed under $\theta$. But then $((() : (B \& C)\theta))$ will succeed as well, since the success of one closed formula cannot affect the success of another.

$S(\lor)$: $B \lor C$ succeeds under $\theta$ iff either $B$ or $C$ succeed under $\theta$, iff $S(B) \lor S(C)$ is valid under $\theta$.

$S(\exists)$, left: If $(\exists x \ B)\theta$ succeeds, there is (by Theorem 2.5.5) some $t$ such that $(B[x := t])\theta$ succeeds, so $(\exists x S(B))\theta$ is valid under $\theta$.

$S(\exists)$, right: From the “one-succeeds-succeeds” theorem (Theorem 2.5.7).

$F(=)$: $F(s = t)$ is valid under $\theta$ iff $s\theta$ and $t\theta$ do not unify; but since $s\theta$ and $t\theta$ are both closed, this happens iff they are non-identical, that is, iff $s = t$ is invalid (and thus $\neg s = t$ is valid) under $\theta$.

$F(\&)$, left: If $B \& C$ fails under $\theta$, it must be the case that $((() : B\theta, C\theta))$ fails. But since the two formulae share no free variables (because both are closed under $\theta$), this means that one of the two fails independent of the other. So either $B$ or $C$ must fail under $\theta$, and $F(B) \lor F(C)$ is valid under $\theta$. (The same holds if we replace “failure” by “flat failure”.)

$F(\&)$, right: If $F(B) \lor F(C)$ is valid under $\theta$, then either $B\theta$ or $C\theta$ must fail; but then $((() : B\theta, C\theta))$ must fail too, so $F(B \& C)\theta$ is valid. (The same holds if we replace “failure” by “failure without predicate calls”.)

$F(\lor)$: $B \lor C$ fails under $\theta$ iff both $B$ and $C$ fail under $\theta$, iff $F(B) \& F(C)$ is valid under $\theta$. (The same holds if we replace “failure” by “flat failure”.)

$F(\exists)$, left: There are two subcases, one for $F^Y$ and another for $F^N$. If $\exists x \ B$ fails under $\theta$, then (by Theorem 2.5.2) $B[x := t]$ fails under $\theta$, for every $t$; so $\forall x F^Y(B)$ is valid under $\theta$. If $\exists x \ B$ fails flatly under $\theta$, then (by reasoning similar
to that in the \( F(\exists), \) right case) \( \mathcal{B}[x := t] \) fails flatly under \( \theta \); so \( \forall x \ F^N(\mathcal{B}) \) is valid under \( \theta \).

\( F(\exists), \) right: If \( \forall x \ F^N(\mathcal{B}) \) is valid under \( \theta \), then for all \( t \), the backtrack stack \((() : \mathcal{B}[x := t]\theta)\) can fail flatly. Consider the backtrack stack \((() : \mathcal{B}'[x := t]\theta\), where \( \mathcal{B}' \) is \( \mathcal{B} \) with all predicate calls replaced by \textit{true}. This backtrack stack can also fail flatly; therefore it cannot succeed (otherwise it would contradict the Church-Rosser property).

So \((() : \mathcal{B}'[x := t]\theta)\) fails for every \( t \); but then the backtrack stack \((() : (\exists x \mathcal{B}')\theta)\) cannot succeed (otherwise it would contradict Theorem 2.5.5), and it cannot diverge (since it has no predicate calls), so it must fail. Now, the new \textit{true} (i.e., \( 0 = 0 \)) subformulae in \( \mathcal{B}' \) cannot have any effect on the failure of this backtrack stack, since they have no effect on the substitution in any closure; therefore there is a failing computation which does not perform Unification steps on the \textit{true} subformulae; therefore if these subformulae are replaced by anything, we still get a failing backtrack stack. In particular, the backtrack stack \((() : (\exists x \mathcal{B})\theta)\) must fail flatly.

\( F^N/F^Y \): trivial.

Unf: By Theorem 3.3.2.

\( F^N(\mathcal{P}) \): The formula \( \mathcal{P}(t_1, \ldots, t_n) \) clearly cannot fail without performing Defined Predicate steps, so the sequent as a whole is valid.

\( \Box \)

\textbf{Theorem 3.3.4 (Soundness of LKE+PAR)} Every sequent derivable in LKE+PAR is valid.

\textbf{Proof.} By induction on the structure of the derivation. Cases are from the Validity theorems for LKE and PAR.

\( \Box \)
3.4 Completeness: Closed Assertions

The kinds of completeness results which we can prove of such systems as LKE+PAR tend to fall into groups based on the general approach taken in the proof. One such group is centred around the idea of restricting the places in which free variables can occur, and is the group which we will turn our attention to in this section.

The results in this section and the next are split into a number of stages. This is not because each stage necessarily states an important result, but rather because each stage uses induction over a different measure.

3.4.1 Completeness for Equality Sequents

I will begin this sequence of results by proving the completeness of LKE for sequents containing only equality formulae. This result will be used in several sections to follow.

**Lemma 3.4.1** Every sequent \( S \) of the form \( \rightarrow s_1 = t_1, \ldots, s_m = t_m \), such that there is no \( i \) such that \( s_i \equiv t_i \), is invalid.

**Proof.** By induction on the measure \( j \cdot \omega + k \), where \( j \) is the number of free variables in the sequent and \( k \) is the number of occurrences of function symbols. When the measure is zero, the sequent is empty and thus invalid. When it is non-zero, we have cases on the form of \( S \).

If there is a formula of the form \( f(\ldots) = g(\ldots) \) in \( S \), then let \( S' \) be \( S \) without this formula. By the induction hypothesis, \( S' \) is invalid; that is, there is a substitution \( \theta \) under which all the equalities in \( S' \) are invalid. But since \( f(\ldots) = g(\ldots) \) is invalid under any substitution, all the equalities in \( S \) are invalid under \( \theta \) as well.

Otherwise, if there is a formula of the form \( f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \), then since the sequent is invalid, the two terms in the equality must not be identical;
so there must be an \( i \) such that \( s_i \neq t_i \). Let \( S' \) be the sequent with this formula replaced by \( s_i = t_i \). By the induction hypothesis, there is a \( \theta \) under which all the formulae in \( S' \) are invalid; but then \( f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \) is clearly invalid under \( \theta \) as well, and \( S \) is thus also invalid.

Otherwise, all formulae in \( S \) are of the form \( x = t \) (or its inverse). Consider the variable \( x \) in the first equality. Let \( s_1, s_2, s_3, \ldots \) be an infinite sequence of closed terms, and let \( i \geq 1 \) be such that \( x = s_i \) does not appear in \( S \). (Note the use of the assumption that we have an infinite number of closed terms; see the discussion at the end of the chapter.) Now let \( S' \) be \( S \) with all occurrences of \( x \) replaced by \( s_i \). \( S' \) has one fewer free variable than \( S \), so by the induction hypothesis, \( S' \) is invalid; that is, there is a substitution \( \theta \) under which all formulae of \( S' \) are invalid. But this amounts to saying that all formulae of \( S \) are invalid under the substitution \( [x := s_i]\theta \).

\( \Box \)

For the next theorem, we need a short technical lemma to make clear that equality is essentially symmetric under the rules for equality we have given in LKE.

**Lemma 3.4.2 (Symmetry of Equality)** The sequent \( s = t \leftrightarrow t = s \) is derivable in LKE.

**Proof.** If we take \( A(x) \) to be \( x = s \), we can use the Sub,r rule of LKE to derive the sequent:

\[
\begin{align*}
 s = t &\rightarrow A(s) \\
 s = t &\rightarrow A(t) \quad i.e. \quad s = t \leftrightarrow s = s \\
 s = t &\rightarrow t = s
\end{align*}
\]

\( \Box \)

**Theorem 3.4.3 (Completeness for Equality)** All valid sequents \( S \) containing only equality formulae are derivable in LKE.
Chapter 3. Characterising Parallel Systems

Proof. By induction on the measure $j \cdot \omega + k$, where $j$ is the number of free variables in the sequent and $k$ is the number of occurrences of function symbols. When the measure is zero, the sequent is empty and thus invalid, so the result holds trivially.

If there are formulae of the form $f(s_1, \ldots, s_p) = f(t_1, \ldots, t_p)$ in the antecedent, then we can use the Comp rule to produce a sequent which is also valid, but with a lower measure (some of the occurrences of function symbols are gone). By the induction hypothesis, this must be derivable.

Otherwise, if $S$ is of the form of an Ineq or Occ axiom of LKE, it is clearly derivable.

Otherwise, if there are any formulae in the antecedent at all, they must be of the form $x_p = t_p$ (or its inverse – see the Lemma), where $x_p$ does not occur in $t_p$. Consider the sequent $S'$ made from $S$ by replacing all other occurrences of $x_1$ by $t_1$ by running the Sub,l and Sub,r rules backwards, and eliminating $x_1 = t_1$ by running the Thin,l rule backwards. Now, if $S'$ were invalid, it would be because all the antecedent formulae were valid under some $\theta$, but some consequent formula was not. But then $S$ would be invalid under the substitution $[x := t]_\theta$ – because the first equality would be valid and the rest of $S$ would be the same as $S'$ under $\theta$ – contradicting our assumption; so $S'$ is also valid. $S'$ is of lower measure than $S$ because although it may have more occurrences of function symbols, it has one fewer free variable; so by the induction hypothesis, it must be derivable. Since we have constructed $S'$ by running rules backwards from $S$, $S$ is also derivable.

Otherwise, there is an empty antecedent. If there are formulae $s = s$ in the consequent, then the sequent is an instance of the Eq axiom from LKE.

Otherwise, by the lemma, the sequent cannot be valid, so the result holds trivially.

$\Box$
Chapter 3. Characterising Parallel Systems

3.4.2 Closed Completeness

Theorem 3.4.4 (Closed Completeness, stage 1) All valid sequents $S$ which have only equalities in the antecedent, and only equalities and $F^N$ formulae in the consequent, are derivable in LKE+PAR.

Proof. By induction on the number of connectives and equality formulae within $F^N$ signs. If this number is 0, then by the Completeness Theorem for Equality, Theorem 3.4.3, the result holds.

Otherwise, let $S$ be $\Gamma \rightarrow \Delta, F^N(D_1), \ldots, F^N(D_m)$, where the $\Delta$ formulae are all equalities. Cases are on $D_1$. We will derive sequents which must also be valid, and which have fewer connectives and equality formulae within $F^N$ signs.

$D_1 \equiv (s = t):$ Assume that $S$ is valid, and let $S'$ be $S$ with $F^N(s = t)$ taken out of the consequent, and $s = t$ put into the antecedent. If $S'$ were invalid, it would be because there is a $\theta$ such that $s\theta \equiv t\theta$ but none of $F^N(D_2)\theta \ldots F^N(D_m)\theta$ were valid. But then $F^N(s = t)$ would not be valid under $\theta$ either, so $S$ would be invalid; contradiction. So $S'$ must be valid; by the induction hypothesis, it is derivable; and $S$ can be derived from it by an application of the $\neg, r$ rule and an application of Cut with the $F(=)$, right axiom from PAR.

$D_1 \equiv B \& C$: Assume that $S$ is valid, and let $S'$ be $S$ with $F^N(B \& C)$ replaced by $F^N(B), F^N(C)$. As in the proof of the validity of the $F(\&)$ axioms, $S'$ must be valid, and $S$ can be derived from it by $\lor, r$ and Cut with $F(\&)$, right.

$D_1 \equiv B \lor C$: Similar to the last case; we can use $\&, r$ and $F(\lor)$, right from PAR to derive $S$ from two valid sequents of lower measure.

$D_1 \equiv \exists x B$: Similar to the last two cases; we can use $\lor, r$ and $F(\exists)$, right from PAR to derive $S$ from a valid sequent of lower measure.

Note that $D_1$ cannot be a predicate application formula because then its failure would inevitably involve a Defined Predicate step.

$\Box$
In the next stage, we will show completeness for $F_Y$ signed formulae, by showing that we can unfold the formula and then use the $F^N/F_Y$ rule. For this, we need to have the following lemma.

**Lemma 3.4.5** If a backtrack stack $\beta$ fails, then some predicate unfolding of $\beta$ can fail without performing any Defined Predicate steps.

**Proof.** By induction on the number of Defined Predicate steps in the failing computation of $\beta$. If this is zero, we already have the result. If it is more than zero, then consider the first Defined Predicate step. The indicated predicate call must be an occurrence of a predicate call which appears in $\beta$; let $\beta'$ be $\beta$ with that occurrence unfolded. $\beta'$ has a computation which is identical to that of $\beta$, except for one subformula being different and at least one Defined Predicate step being missing (the first one, and possibly later ones resulting from copies of the predicate call being generated by the $\lor$ rule). The result follows from the induction hypothesis.

\[ \square \]

**Theorem 3.4.6 (Closed Completeness, stage 2a)** All valid sequents $S$ of the form $[\neg \rightarrow F_Y(A)]$, where $A$ is a closed formula, are derivable in LKE+PAR.

**Proof.** $A$ is a closed formula with a failing computation, so the backtrack stack $((A : A))$ fails. By the Lemma, there is therefore some unfolding $A'$ of $A$ such that $((A : A'))$ fails without Defined Predicate steps. The sequent $[\neg \rightarrow F^N(A')]$ is therefore valid, and (by the last stage) therefore derivable; $[\neg \rightarrow F_Y(A)]$ is derivable from it by one application of Cut with $F^N/F_Y$ of PAR and zero or more applications of Unf, right of PAR.

\[ \square \]

Note that this last proof would not have worked if $A$ had any free variables, since the lemma does not guarantee that there will be a single unfolding which fails without predicate calls for *all* instances of $A$. 

Theorem 3.4.7 (Closed Completeness, stage 2b) All valid sequents $S$ of the form $[\to S(A)]$, where $A$ is a closed formula, are derivable in LKE+PAR.

Proof. By the definition of validity, $(\top : A)$ must succeed. We can proceed by induction on the length of this computation. Cases are on the form of $A$.

$A \equiv (s = t)$: $s$ and $t$ are closed, and so must be identical for them to unify. We can therefore derive $S$ from an Eq axiom and an application of Cut with the $S(=)$, right axiom of PAR.

$A \equiv B \& C$: $B$ and $C$ must succeed independently, each with a shorter computation than $B \& C$. $S$ must therefore be derivable from $[\to S(B)]$ and $[\to S(C)]$ by $\&$,r and Cut with $S(\&)$, right of PAR.

$A \equiv B \lor C$: The first step of the successful computation of $B \lor C$ must be to split the closure into two on the backtrack stack. By Lemma 2.3.4, either the backtrack stack $(\top : B)$ or the backtrack stack $(\top : C)$ must succeed; therefore, by the induction hypothesis either $[\to S(B)]$ or $[\to S(C)]$ must be derivable. We can derive $S$ from the derivable one by applications of Thin,r and $\lor$,r and Cut with $S(\lor)$, right from PAR.

$A \equiv \exists x B$: By the "succeeds-one-succeeds" theorem (Theorem 2.5.5), there must be a closed $t$ such that $B[x := t]$ succeeds, and it must do so with a computation with at least one step fewer than that of $\exists x B$. By the induction hypothesis, therefore, $[\to S(B[x := t])]$ must be derivable; and $S$ is derivable from it by $\exists$,r and Cut with $S(\exists)$, right of PAR.

$A \equiv P(t_1, \ldots, t_n)$: The successful computation must start with a Defined Predicate step, and so the predicate 1-unfolding of $A$ must have a shorter computation than $A$. $S$ is therefore derivable by Cut with Unf, right of PAR.

□

Theorem 3.4.8 (Closed Completeness, stage 3) All valid sequents $S$ which have only equalities in the antecedent, and only equalities and signed formulae in the consequent, and where no free variable appears in any $S$ or $F^\forall$ assertion, are derivable in LKE+PAR.
Proof. If some $S$ or $F'$ assertion in the consequent is valid, we can derive $S$ by thinning out all other assertions (by stages 2a and 2b). Otherwise, none of the $S$ or $F'$ assertions can be valid under any substitution (since they are closed); so those assertions can be thinned out and the remaining sequent is derivable (by stage 1).

In the last stage, we will generalise the third stage in the direction of permitting arbitrary assertions in sequents. However, we still cannot handle signed formulae in the antecedent at the base of the induction, so we must ban them from any context where they will end up in the antecedent. (But see the discussion on guaranteed-terminating queries, below.) This leads to the standard concept of “positive” and “negative” contexts.

Definition 3.4.9 An occurrence of a term, formula, signed formula, or assertion appears in a positive context in a sequent if it appears within an even number of negations in the consequent, or within an odd number of negations in the antecedent. It appears in a negative context in a sequent if it appears within an odd number of negations in the consequent, or within an even number of negations in the antecedent.

Theorem 3.4.10 (Closed Completeness, stage 4) All valid sequents $S$ in which no free variable appears in an $S$ or $F'$ subassertion, and no signed formula appears in a negative context, are derivable in LKE+PAR.

Proof. By induction on the total number $n$ of connectives outside signed formulae. $n = 0$ is the previous stage. For $n > 0$, we have cases on the first connective; these cases follow straightforwardly from the induction hypothesis, except for the case of an existential quantifier ($\exists x B$) on the left.

In that case, let $S'$ be the premiss of an application of the $\exists,1$ rule of which $S$ is the conclusion. If $S'$ were invalid, it would be because under some $\theta$, all of its antecedent assertions but none of its consequent assertions are valid. But if $B[x := y]\theta$ is valid, then $(\exists x B)\theta$ must be valid (since whatever $\theta$ substitutes for $y$
is a witness for \( x \)); but then \( S \) would be invalid, since under \( \theta \) all of its antecedent assertions and none of its consequent assertions are valid; contradiction. Therefore \( S' \) must be valid, and by the induction hypothesis is derivable; so \( S \) is derivable as well.

\( \square \)

Some examples of sequents which are valid but underivable in LKE+PAR are \( [\rightarrow S(Add(x,0,x))] \), where \( Add \) is the familiar addition predicate for Peano integers (underivable because it has a free variable), and \( [S(Loop()) \rightarrow] \), where \( Loop \) is the infinite loop predicate (valid because the assumption that \( Loop() \) succeeds is false, but underivable because the expansion of \( Loop() \) goes on forever). We will look at ways in which these sources of incompleteness can be handled in Chapter 5.

There are also sequents which are valid and derivable, but not covered by the Closed Completeness results here; one example is \( [F(s = s) \rightarrow] \). This sequent will fall into a class of sequents to be proven derivable in Section 3.5 below, but in general it seems difficult to describe natural and useful classes of valid, derivable sequents. The class described in Closed Completeness, stage 4 and the class described in Section 3.5 are two of the most natural.

Note that because of the restrictions on the class of valid sequents being proved derivable, the left-hand rules of PAR were never used in the completeness proof. They will be used in the proofs in Section 3.5.

### 3.4.3 Characterisation Results

The results in this section avoid the negative results about incompleteness discussed in Chapter 5. However, this incompleteness arises in part because of the great expressive power of sequents. We have already effectively characterised the classes of queries which succeed or fail in a system with parallel connectives. Here, therefore, are the main positive results arising from the completeness theorems; they are some of the most important results in the thesis.
Chapter 3. Characterising Parallel Systems

Theorem 3.4.11 (Characterisation of SOS) A goal formula $A$ succeeds in SOS iff the sequent $[\rightarrow S(\exists[A])]$ is derivable in LKE+PAR; it fails in SOS iff the sequent $[\rightarrow F^Y(\exists[A])]$ is derivable in LKE+PAR.

Proof. First note that $A$ succeeds (fails) iff $\exists[A]$, its existential closure, succeeds (fails); this is because the backtrack stack $(0 : \exists[A])$ must be computed by first doing one $\exists$ step for each quantified variable. The rest of the theorem follows from the Soundness theorem for LKE+PAR and the Closed Completeness, stage 3 theorem.

What this means is that we have succeeded in logically characterising the two outer circles of the Venn diagram, Figure 2–6. A query $A$ is in the outer failure set iff $[\rightarrow F^Y(\exists[A])]$ is derivable, and it is in the outer success set iff $[\rightarrow S(\exists[A])]$ is derivable. Because of the great expressive power of sequents, sequents can express many more things than just the success or failure of individual queries; the incompleteness of LKE+PAR is only in these areas.

LKE+PAR has enabled us to achieve one part of our main goal: the logical characterisation of the sets of queries which succeed or fail in the parallel operational semantics SOS. Because of the completeness properties of SOS/so and SOS/sa, we have also characterised the queries which fail in SOS/so and those which succeed in SOS/sa.

3.5 Completeness: Predicate-Free Assertions

Another group of completeness results for LKE+PAR is centred around the idea of disallowing defined predicate formulae in assertions. This restriction can be lifted if we restrict the program to one having no definitions of recursive or mutually-recursive predicates.

Such restrictions are very severe from a logic programming standpoint. In this group, however, there is no restriction on where signed formulae can appear, and
no restriction on free variables (unlike the group in the last section). This tends
to suggest that, to some degree, it is the interaction between free variables and
recursive predicates which causes the lack of a complete proof system.

Theorem 3.5.1 (Flat Completeness, stage 1) All valid sequents $S$ contain-
ing no signed formulae are derivable in LKE.

Proof. By induction on the number of connectives in $S$. The case 0 is the
Completeness for Equality theorem. When there are connectives, we can eliminate
each one with one application of an LKE rule, to form a sequent which is also valid
but with a lower number of connectives. Cases are straightforward, except for the
$\exists！l$ case, which is similar to the reasoning in the Closed Completeness, stage 4
theorem.

Theorem 3.5.2 (Flat Completeness, stage 2) All valid sequents $S$ contain-
ing no predicate calls are derivable in LKE+PAR.

Proof. By induction on $j \cdot \omega + k$, where $j$ is the total number of connectives
within signed formulae, and $k$ is the total number of connectives anywhere in
assertions containing signed formulae. Case 0 is the previous stage (no assertions
contain signed formulae). If there are assertions containing signed formulae, we
have cases on the first such formula in the sequent.

If this formula is not itself a signed formula, we can eliminate its top-level
connective in the manner of the last stage, decreasing $k$ by one. (In the rest of the
cases, we will push one connective outside the sign of a signed formula, leaving $k$
the same but decreasing $j$ by one.)

Otherwise, it is a signed formula. If it is of the form $S(s = t)$, then let $S'$ be $S$
with this assertion replaced by $s = t$. By the validity of the PAR axiom $S(=),l$, $S'$
is valid; we can derive $S$ from it by an application of Cut with $S(=),r$. The same
reasoning holds for the other $S(A)$ assertions, $F(=), F(\&), F(V)$, and $F^N(\exists)$. It
does not hold for $F^V(\exists)$ because there is a different rule for each direction.
Chapter 3. Characterising Parallel Systems

If the signed formula is of the form \( F^Y (\exists x \Box) \) and is in the antecedent, then let \( S' \) be \( S \) with this assertion replaced by \( \forall x F^Y (\Box) \). \( S \) can be derived from \( S' \) by Cut with \( F(\exists), l \); we now need only to show that \( S' \) is valid. Let \( S'' \) be \( S \) with \( F^Y (\exists x \Box) \) replaced by \( \forall x F^N (\Box) \). \( S'' \) is derivable from \( S \) (by Cut with \( F(\exists), r \)), and so must be valid; but since \( \Box \) contains no predicate calls, \( F^N (\Box) \) implies that \( F^N (\Box) \), and \( S' \) is also valid.

If the signed formula is of the form \( F^Y (\exists x \Box) \) and is in the consequent, then let \( S' \) be \( S \) with this assertion replaced by \( \forall x F^N (\Box) \). \( S \) can be derived from \( S' \) by Cut with \( F(\exists), r \) and \( F^N / F^Y \) from PAR; but \( S' \) can be derived from \( S \) by Cut with \( F(\exists), l \), and so must be valid.

This last stage says that as long as we restrict our attention to only predicate-free sequents, we can prove any valid sequent. This is not very useful for characterising logic programming, since there we are mostly concerned with proving properties of recursive predicates. Its utility is mainly in casting light on the question of why there is no complete proof system.

One way of strengthening this result is to allow predicate calls, but to disallow recursion in the base program. This kind of result suggests that it is not exactly predicate calls, but calls to recursive predicates that causes the problem.

Definition 3.5.3 A hierarchical program \( \Pi \) is one in which each predicate \( P \) in the language \( \mathcal{L} \) can be assigned a natural number \( n_P > 0 \) such that if \( P \) calls \( Q \) as defined in \( \Pi \), \( n_Q < n_P \).

Theorem 3.5.4 If \( \Pi \) is hierarchical, then all valid sequents \( S \) are derivable in LKE+PAR.

Proof. By the techniques of this section, except that signed formulae containing predicate calls are expanded by a finite number of applications of Cut with Unf from PAR.

\( \Box \)
There are various ways in which we could combine the results of the last section with those of this section, to describe wider and wider classes of sequents or programs for which LKE+PAR is complete. Unfortunately, these classes are somewhat unnatural and difficult to describe. Here are some brief examples.

**Theorem 3.5.5** All valid sequents in which no predicate calls appear in a negative context, and no \( S \) or \( F' \) formula in a positive context contains both predicate calls and free variables, are derivable in LKE+PAR.

**Proof.** We can eliminate \( S \) and \( F' \) signed formulae containing no predicate calls by the method of the last theorem. After that, the sequent will be in the form of the Closed Completeness, stage 4 theorem of the last section.

Another example is that we could replace the restriction to hierarchical programs by a restriction that only hierarchical predicates within programs can appear in the antecedent. This is not much of a relaxation, because typically, many of the most useful predicates in programs are recursive.

### 3.6 Discussion

Here I present some notes of a more philosophical and historical nature on the contents of this chapter, which would have seemed out of place in the technical discussions earlier.

#### 3.6.1 LKE

Equality has been examined in the context of first order logic for decades. Commentators such as Fitch [Fit52] and Church [Chu56] give axiomatisations for simple equality between first order terms, and note that the axiom \( \forall x(x = x) \) and some form of substitution axiom schemata are sufficient to deduce the transitivity and symmetry of equality. Takeuti [Tak87] casts this into sequent calculus form by
augmenting Gentzen's LK [Gen69] to a calculus $\text{LK}_e$; this calculus has the property that a sequent is derivable in it iff the sequent with the equality axioms added to the antecedent is derivable.

The notion of validity associated with these formalisations of equality leaves open the possibility that two non-identical closed terms might be equal. For the purposes of logic programming, we want to exclude this possibility. We thus want an axiomatisation equivalent to Clark's equality theory [Cla79]; actually, any sequent calculus which has the completeness property for equality between closed terms as syntactic identity, will do for this purpose. $\text{LKE}$ is simply Gentzen's LK with such additional axioms and rules.

Readers might be curious about the classical nature of $\text{LKE}$. Recent results [MNS87] show that logic programming has a strong connection with intuitionistic logic, but $\text{LKE}$ is clearly a classical sequent calculus. However, these results are about the internal logic within logic programming, and in $\text{LKE+PAR}$ we are concerned with notions of proving properties of success and failure of queries in a program logic.

Constructivist principles should not bar us from working classically in this setting, because given our definition of validity, the law of excluded middle clearly holds for all assertions. Note, for instance, that no predicates other than equality can appear at the top level of sequents, and that all other predicate applications are enclosed within signs.

### 3.6.2 PAR

The classicality of the assertion connectives is exploited in the PAR axioms, which present in a clear and concise manner the relationship between goal formula connectives within signed formulae and assertion connectives outside signed formulae. The $S$ axioms state that success of a query is essentially the same as its provability in $\text{LKE}$ (given the expansion of predicates with the Unfolding rule). The $F$ axioms, like De Morgan's laws, state how failure can be "pushed down through" the
Chapter 3. Characterising Parallel Systems

goal formula connectives, converting conjunctions into disjunctions, disjunctions into conjunctions, and existentials into universals.

The only elements which mar the compositionality of the PAR axioms are the $P^N/P^Y$ dichotomy and the associated unfolding of predicate calls within goal formulae. This device is necessitated by the inherent asymmetry in logic programming discussed in Chapter 2: essentially, we terminate if only one solution is found, but rather than terminating if only one counterexample is found, we keep discarding counterexamples until we are satisfied that every element of the domain is a counterexample.

3.6.3 Characterising Success Alone

If we had not wanted to characterise failure in the calculus, we would have been able to simplify it considerably. One simple presentation of the calculus is as follows.

Let predicate application formulae also be assertions. Let the definition of validity with respect to II make no mention of failure, but include a clause to the effect that closed formulae of the form $P(t_1, \ldots, t_n)$ are valid if the corresponding $A(t_1, \ldots, t)$ formula is valid.

Let the axioms PAR' be the two axioms $S(A) \rightarrow A$ and $A \rightarrow S(A)$. We can prove every closed instance of these axioms valid by induction on the length of the computation of $A$ and on the stage of the inductive definition of validity at which $A$ becomes valid. (In fact, in this formulation, the sign $S$ has no significance other than to mark a goal formula, and can be done away with.)

Now consider the set of axioms DEF II formed from the program II, and consisting of an axiom of the form $P(t_1, \ldots, t_n) \rightarrow A(t_1, \ldots, t_n)$ and one of the form $A(t_1, \ldots, t_n) \rightarrow P(t_1, \ldots, t_n)$ for every predicate defined in II. These axioms are trivially valid.

The Closed Completeness and Flat Completeness results of this chapter now hold for LKE+DEF II+PAR': we can replace $S(A)$ by $A$ in any sequent to obtain
a sequent which is also valid, and LKE+DEFn is complete for the new definition of validity.

This line of development is similar in spirit to that of Hagiya and Sakurai [HS84] and Hallnäs and Schroeder-Heister [HSH88]. When we move to expand this calculus to take account of failure, however, we need the concepts of flat failure and predicate unfolding. The predicate unfolding rule extends easily to successful formulae, and thus does away with the need to take account of predicate calls at the top level of sequents.

As we will see in the next chapter, in the context of sequential computation, we need to characterise failure in order to characterise success, so the two signs are necessary from the start. The presentation of LKE+PAR in this chapter is therefore more in harmony with the calculus in the next chapter than is the simpler characterisation I just gave.

### 3.6.4 More Practical Failure Rules

Implicit in the completeness theorems is a method for finding a derivation for \( \rightarrow F^V(\exists x B) \) sequents: unfold occurrences of predicate calls in \( B \) to the required depth, and then use the other rules. This is sufficient for the purpose of proving the theorems, but might be too clumsy for practical use.

An alternative formulation would have the \( F^N \) sign replaced by an infinite number of signs \( F^n \), where \( n \geq 0 \) is the number of Defined Predicate steps allowed in the failing computation. The definition of validity would be changed accordingly. We would then have that \( \forall x F^n(\exists x B) \rightarrow F^V(\exists x B) \) for any \( n \), and the Unfolding axioms could simply take the following forms:

\[
\sigma(A(t_1, \ldots, t_n)) \rightarrow \sigma(P(t_1, \ldots, t_n)); \sigma(P(t_1, \ldots, t_n)) \rightarrow \sigma(A(t_1, \ldots, t_n))
\]

where \( \sigma \) is \( S \) or \( F^V \), and \( P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n) \) is in the program \( \Pi \);

\[
F^n(A(t_1, \ldots, t_n)) \rightarrow F^{n+1}(P(t_1, \ldots, t_n)), \quad F^{n+1}(P(t_1, \ldots, t_n)) \rightarrow F^n(A(t_1, \ldots, t_n))
\]

where \( P(x_1, \ldots, x_n) \leftrightarrow A(x_1, \ldots, x_n) \) is in the program \( \Pi \).

We would then be able to delay the unfolding of predicate calls later, until it
became clear that the unfolding was needed; the choice of $n$ in the application of the $F^n/F^Y$ rule would be left to the user, who might be able to make a good guess on a convenient number. I avoided this formulation only because it would have unnecessarily complicated some of the proofs, for no mathematical gain.

A related idea is to have the validity of an assertion be relative to a fixed bound on how many predicate expansions an $F^N$ formula is allowed to make. We would define $k$-validity of closed assertions as identical to validity, except that $F^N(A)$ would be $k$-valid iff $A$ failed performing $k$ predicate expansions or fewer. We would then define validity for sequents as follows: a sequent is valid iff there is a $k$ such that for all $\theta$, if all the antecedent formulae are valid under $\theta$, then one of the consequent formulae is valid under $\theta$. The PAR rules would change as in the last paragraph. Again, although this might be more practically convenient, I have avoided this formulation due to its unnecessary complexity.
Chapter 4

Characterising Sequential Systems

In this chapter, I will give a characterisation of the two inner circles of the Venn diagram in Figure 2–6 in the same way as I characterised the two outer circles. That is, I will give a proof-theoretic characterisation of sequential logic programming (in particular, the operational semantics SP) in the form of a sequent calculus.

For this sequent calculus, we can use the rules LKE from the last chapter unchanged; we need only give a new group of axioms, SEQ, corresponding to PAR from the last chapter. These axioms, however, are more complex than those in PAR, have more side-conditions, and in particular involve the concept of disjunctive unfoldings of formulae.

Nevertheless, we can prove the same things about SEQ that we can about PAR: the laws are sound, and the proof system LKE+SEQ characterises sequential logic programming in several useful ways.

I will also give a characterisation of the last circle in Figure 2–6, namely the middle success circle. This set contains all queries which succeed in SOS/so, and can be characterised by a set of axioms, PASO, which combines axioms from PAR and from SEQ in a simple and intuitively clear way.
4.1 Approaches to Semantics

I begin by going into more detail about why we want a semantics for sequential logic programming, and what approaches have been taken so far to giving one.

The assumptions made about search strategies in most research on foundations of logic programming (for instance, SLD-resolution with a fair search rule) are not satisfied by sequential logic programming. Sequential Prolog systems may diverge in cases where fair SLD-resolution can succeed, or in cases where parallel Prologs can fail.

However, it seems clear that sequential Prolog is a useful language – and thus needs a mathematical semantics which will allow us to do such things as proving termination and correctness properties of programs. Various approaches have been taken to describing termination and correctness, including analyses of the operational semantics, and denotational descriptions that implicitly take termination into account.

4.1.1 Operational Approaches

In a paper of Francez et al. [FGKP85], a characterisation is given of terminating Prolog computations in terms of operational semantics. One can prove that a computation terminates by giving conditions on the form of the tree of candidate solutions: if there are no infinite branches to the left of the first solution, then the program terminates. Francez et al. also give a proof system in which proofs of properties of programs can be made.

This is an adequate method of characterising termination. However, the operational semantics of a logic programming language is clearly secondary to the declarative semantics, which is where the whole purpose of the language comes from. A characterisation of termination in terms of the underlying logic of the language would be preferable to this purely operational description. Their proof system approach, while having a logical structure, reifies such concepts as answer
substitutions and unification, which are more properly of the operational semantics than the abstract logical structure of Prolog programs. We therefore achieve very little abstract mathematical insight from this technique.

4.1.2 Denotational Approaches

There have been several denotational analyses of sequential Prolog (some examples are [JM84,AB87,DM88,Bau88,dBdV89,NF89]), and these seem to bring out deeper and more abstract properties of the language.

Of these, one appealing example is Baudinet's [Bau88]. In this semantics, queries are given denotations in the set of functions which map substitutions to (essentially) sequences of substitutions. "Append" and "join" operators are defined which allow the derivation of denotations of composite goals, and of sequences of clauses, from the denotations of individual atoms. This semantics is therefore compositional and has clear connections to standard techniques in denotational semantics.

However, it is disappointing that this characterisation is so functional and so far removed from logic. Unification is again reified, and the result of a computation is viewed as the result of a composition of functions applied to the empty substitution. One might argue that this kind of functional denotational semantics is inappropriate for logic programming for philosophical reasons: denotational semantics views every program as a function from inputs to outputs, while the whole point of the logic programming paradigm is that it is not necessary to view a program in this way. The reason that programmers choose a logic programming language is often that their problem has some inherent logical structure which is reflected in the language, and not well described by functions.

One advantage of denotational semantics approaches like Baudinet's is that they allow a description of cut and negation as failure which is well-integrated with the rest of the semantics. The well-known "non-logical" nature of these features remains, however.
Chapter 4. Characterising Sequential Systems

4.2 Disjunctive Unfoldings

The notion of the disjunctive unfolding of a formula is one of the main novelties of this thesis, and the mechanism which allows us to isolate the non-compositionality of sequential Prolog. The disjunctive unfolding of a formula $A$ is a formula $A'$ which is classically equivalent to $A$, but has the property that its satisfiability depends only on the satisfiability of its subformulæ.

This requires some motivation. Once we have set out to develop a proof system characterising sequential Prolog, there is one fairly natural way to proceed (which has been followed independently, for example, by Girard [Gir87]). However, the resulting proof system still has soundness problems; as with full first order logic, we can still prove things which have no corresponding computation.

Unfoldings of formulæ are exactly what we need to solve these soundness problems. This section will present the idea of unfoldings by giving an outline of the initial attempt at a proof system, describing that system’s problems, defining the predicate and disjunctive unfoldings of a formula, and proving some essential properties of unfoldings.

4.2.1 An Initial Attempt at a Characterisation

We would like to give axioms which characterise sequential logic programming as the PAR axioms characterise parallel logic programming. For instance, in sequential logic programming, $B\&C$ succeeds if both $B$ and $C$ succeed, and $B\&C$ fails if either $B$ fails, or $B$ succeeds and $C$ fails. We might imagine that the axioms for success and failure of conjunctions would therefore be something like the following:

\[
\begin{align*}
S(\&): & \quad S(B\&C) \leftrightarrow S(B)\&S(C) \\
F(\&): & \quad F(B\&C) \leftrightarrow F(B) \lor (S(B)\&F(C))
\end{align*}
\]

The axioms for success and failure of disjunctions would presumably be the duals
of these:

\[
S(V) \vdash S(B \lor C) \leftrightarrow S(B) \lor (F(B) \land S(C)) \quad \text{and} \quad F(V) \vdash F(B \lor C) \leftrightarrow F(B) \land F(C)
\]

However, although the \( \rightarrow \) direction of these axioms are sound, the \( \leftarrow \) direction of the \( F(\&) \) axiom – the direction we need to prove sequents of the form \([ \rightarrow F(B \& C)] \) – is not sound. Consider the query \((true \lor Loop()) \& false\). This query diverges according to the operational semantics \(SP\); the transitions are

\[
((): (true \lor Loop()) \& false) \xrightarrow{SP} ((): true \lor Loop(), false) \xrightarrow{SP} \\
((): true, false);((): Loop(), false) \xrightarrow{SP} ((): false);((): Loop(), false) \xrightarrow{SP} \\
((): Loop(), false) \xrightarrow{SP} ((): Loop(), false) \xrightarrow{SP} \ldots
\]

However, with the rules given above, we can “prove” that it fails.

\[
\begin{align*}
& \rightarrow 0 = 0, F(true) \& S(Loop()) \\
& \rightarrow S(true), F(true) \& S(Loop()) \\
& 0 = 1 \rightarrow \\
& \rightarrow S(true \lor Loop()) \rightarrow \neg(0 = 1) \\
& \rightarrow S(true \lor Loop()) \rightarrow F(false) \\
& \rightarrow F(true \lor Loop()), S(true \lor Loop()) \& F(false) \\
& \rightarrow F(true \lor Loop()) \lor (S(true \lor Loop()) \& F(false)) \\
& \rightarrow F((true \lor Loop()) \& false)
\end{align*}
\]

But now consider the query \((true \& false) \lor (Loop() \& false)\). This is classically equivalent to the previous query, and is handled in much the same way by the operational semantics \(SP\); in fact, the computation is exactly the same after one steps. However, we cannot get a derivation of \([ \rightarrow F((true \& false) \lor (Loop() \& false))] \); in other words, the proof system behaves correctly in regard to this query.

Basically, if a query has all its disjunctions outside all its conjunctions and existential quantifiers, then it will be handled correctly by the simple proof rules above. Many queries will not have this property. However, a query can be transformed in this direction by taking the first disjunction encountered in a depth-first traversal of the formula, and “pulling it toward the outside of the formula” by distributing the conjunctions and quantifiers over it. There is the added compli-
Chapter 4. Characterising Sequential Systems

Figure 4–1: $B$ is the key subformula of formula $A$. Informally, the shaded region consists only of $=$, $\&$, and $\exists$ formulae.

Figure 4–2: Disjunctive unfolding of a formula. The indicated disjunction, $B \lor C$, is the key subformula of $A$.

4.2.2 Definitions and Examples

The above discussion motivates the following definitions.

Definition 4.2.1 The key subformula of a formula is the leftmost disjunction or predicate application subformula which is not a proper subformula of another disjunction. (See Figure 4–1.) Not every formula has a key subformula.

The disjunctive unfolding of a formula $A$ with key subformula $B \lor C$ is the formula $A^B \lor A^C$, where $A^B$ (resp. $A^C$) means $A$ with its key subformula replaced by $B$ (resp. $C$). (See Figure 4–2.) Formulae which do not have a disjunctive key subformula have no disjunctive unfolding; we write $A \xrightarrow{\text{Disj}} A'$ if $A'$ is the disjunctive unfolding of $A$. 

Examples. The formulae $(B \lor C) \& s = t, \exists x (s = t \& (B \lor C))$, and $B \lor C$ all have $B \lor C$ as their key subformulae. The disjunctive unfoldings of these formulae are, respectively, $(B \& s = t) \lor (C \& s = t); \exists x (s = t \& B) \lor \exists x (s = t \& C);$ and $B \lor C$ itself. The formula $P(x) \& (B \lor C)$ has $P(x)$ as its key subformula; the formula $\exists x (x = 0 \& x = 1)$ has no key subformula.

As we will see in future sections, we can make the simple proof rules given in section 4.2.1 work properly if we first restrict certain formulae in the rules to be formulae without predicate application subformulae, and then add a rule allowing a formula to be transformed by a disjunctive unfolding as well as a predicate unfolding.

4.2.3 Some Facts about Unfoldings

For now, we must show that the disjunctive unfolding of a formula, like predicate unfoldings, are equivalent to that formula under the operational semantics. This fact will be important for proving the soundness and completeness results to come. Unfortunately, this proof is not as straightforward as that for predicate unfoldings, and involves a long case analysis.

Theorem 4.2.2 (Operational Equivalence of Disjunctive Unfoldings)
If $A \xrightarrow{\text{Dij}} A'$, then $(\theta : A, \alpha);\beta$ succeeds (fails) in SP iff $(\theta : A', \alpha);\beta$ succeeds (fails) in SP.

Proof. $(\rightarrow)$ By induction on the depth of the key subformula (which is a disjunction) in the tree of the formula $A$. In what follows, I use the notation $\beta \xrightarrow{\text{SP}} S/F$ to denote that $\beta$ succeeds or fails, its meaning being consistent through the analysis of each case.

Case depth = 0: $A$ is itself a disjunction, so its disjunctive unfolding is itself. The result holds trivially.

Case depth > 0: $A$ can be either a conjunction or an existential formula.
A ≡ B&C. We have the computation
\[(\theta : B&C, \alpha); \beta \xrightarrow{SF} (\theta : B, \alpha); \beta \xrightarrow{SF} S/F\]

There are two subcases to consider.

- The key subformula, \(B_1 \lor B_2\), is in \(B\). The disjunctive unfolding of \(B\) is therefore \(B_1 \lor B_2\), and \(A'\) is \((B_1 \land B_2) \lor (B_2 \land C)\). By the induction hypothesis, we have that
\[(\theta : (B_1 \lor B_2, C, \alpha); \beta \xrightarrow{SF} (\theta : B_1, C, \alpha); (\theta : B_2, C, \alpha); \beta \xrightarrow{SF} S/F\]
But then
\[(\theta : (B_1 \land C) \lor (B_2 \land C), \alpha); \beta \xrightarrow{SF} (\theta : B_1, C, \alpha); (\theta : B_2, C, \alpha); \beta \xrightarrow{SF} S/F\]
because we can insert the step for splitting the second conjunction at the appropriate point if necessary. So, since \(A' \equiv (B_1 \land C) \lor (B_2 \land C)\), the result holds.

- The key subformula, \(C_1 \lor C_2\), is in \(C\). Thus \(B\) contains no disjunctions or predicate calls, and \(A'\) is \((B \land C_1) \lor (B \land C_2)\). We have that
\[(\theta : B, C, \alpha); \beta \xrightarrow{Sp} (\theta' : C, \alpha); \beta \xrightarrow{Sp} S/F\]
(because \(B\) contains no disjunctions, it cannot lengthen the backtrack stack). Thus, by the induction hypothesis, we have that
\[(\theta' : C_1 \lor C_2, \alpha); \beta \xrightarrow{Sp} (\theta' : C_1, \alpha); (\theta' : C_2, \alpha); \beta \xrightarrow{Sp} S/F\]
But then
\[(\theta : (B \land C_1) \lor (B \land C_2), \alpha); \beta \xrightarrow{Sp} (\theta : B \land C_1, \alpha); (\theta : B \land C_2, \alpha); \beta \xrightarrow{Sp} S/F\]
because we can insert the computation for deriving \(\theta' : C_2, \alpha); \beta\) from \((\theta : B \land C_2, \alpha); \beta\) at the appropriate point if necessary. So, since \(A' \equiv (B \land C_1) \lor (B \land C_2)\), the result holds.
A \equiv \exists x \mathcal{B}. A' is \exists x \mathcal{B}^1 \lor \exists x \mathcal{B}^2. We have the computation

\((\theta : \exists x \mathcal{B}, \alpha); \beta \xrightarrow{SP} (\theta : \mathcal{B}[x := x'], \alpha); \beta \xrightarrow{SP^*} S/F\)

By the induction hypothesis, we have that

\((\theta : \mathcal{B}[x := x']^B_1 \lor \mathcal{B}[x := x']^B_2, \alpha); \beta \xrightarrow{SP} (\theta : \exists x \mathcal{B}^1, \alpha); (\theta : \mathcal{B}[x := x']^B_2, \alpha); \beta \xrightarrow{SP^*} S/F\)

But then

\((\theta : \exists x \mathcal{B}^B_1 \lor \exists x \mathcal{B}^B_2, \alpha); \beta \xrightarrow{SP} (\theta : \exists x \mathcal{B}^B_1, \alpha); (\theta : \exists x \mathcal{B}^B_2, \alpha); \beta \xrightarrow{SP^*} S/F\)

because we can insert the step for discharging the existential quantifier at the appropriate point if necessary (we can even use the same new variable, \(x'\), because if we get to that point, all other occurrences of that variable will have disappeared due to failure). So, since \(A' \equiv \exists x \mathcal{B}^B_1 \lor \exists x \mathcal{B}^B_2\), the result holds.

This completes the proof of the forward direction of the theorem statement.

We now move on to the proof of the converse.

\((\leftarrow)\) By induction on the depth of the key subformula (which is a disjunction) in the tree of the formula \(A\).

Case depth = 0: \(A\) is itself a disjunction, so its disjunctive unfolding is itself. The result holds trivially.

Case depth > 0: \(A\) can be either a conjunction or an existential formula.

- \(A \equiv \mathcal{B} \& \mathcal{C}\). There are two subcases to consider.

  - The key subformula, \(\mathcal{B}_1 \lor \mathcal{B}_2\), is in \(\mathcal{B}\). \(A'\) is therefore \((\mathcal{B}_1 \& \mathcal{C}) \lor (\mathcal{B}_2 \& \mathcal{C})\). We have the computation
    \((\theta : (\mathcal{B}_1 \& \mathcal{C}) \lor (\mathcal{B}_2 \& \mathcal{C}), \alpha); \beta \xrightarrow{SP} (\theta : \mathcal{B}_1 \& \mathcal{C}, \alpha); (\theta : \mathcal{B}_2 \& \mathcal{C}, \alpha); \beta \xrightarrow{SP} \)
But then we must have the computation

$$(\theta : B^{B_1}, C, \alpha); (\theta : B^{B_2} & C, \alpha); \beta \xrightarrow{SF} S/F$$

because the effect of the goal stack $(B^{B_2}, C, \alpha)$ is identical to that of $(B^{B_2} & C, \alpha)$ So by the induction hypothesis, we have that

$$(\theta : B, C, \alpha); \beta \xrightarrow{SF} S/F$$

So, adding one step onto the front of the computation,

$$(\theta : B & C, \alpha); \beta \xrightarrow{SF} S/F$$

The key subformula, $C_1 \lor C_2$, is in $C$. $B$ therefore contains no predicate calls or disjunctions, and $A'$ is $(B & C^{C_1}) \lor (B & C^{C_2})$. We have the computation

$$(\theta : (B & C^{C_1}) \lor (B & C^{C_2}), \alpha); \beta \xrightarrow{SF} (\theta : B & C^{C_1}, \alpha); (\theta : B & C^{C_2}, \alpha); \beta \xrightarrow{SF} (\theta : B, C^{C_1}, \alpha); (\theta : B & C^{C_2}, \alpha); \beta \xrightarrow{SF} S/F$$

(The second-last sequence of steps does not lengthen the backtrack stack because $B$ has no disjunctions or predicate calls.) But then we must have the computation

$$(\theta' : C^{C_1} \lor C^{C_2}, \alpha); \beta \xrightarrow{SF} (\theta' : C^{C_1}, \alpha); (\theta' : C^{C_2}, \alpha); \beta \xrightarrow{SF} S/F$$

because $(\theta : B & C^{C_2}, \alpha) \xrightarrow{SF} (\theta' : C^{C_2}, \alpha)$. So by the induction hypothesis, we have that

$$(\theta' : C, \alpha); \beta \xrightarrow{SF} S/F$$

So, adding some steps onto the front of the computation,

$$(\theta : B & C, \alpha); \beta \xrightarrow{SF} (\theta : B, C, \alpha); \beta \xrightarrow{SF} (\theta' : C, \alpha); \beta \xrightarrow{SF} S/F$$
\[ A \equiv \exists x B. \quad A' = (\exists x B_1) \lor (\exists x B_2). \] We have the computation
\[
(\theta : (\exists x B_1) \lor (\exists x B_2), \alpha) ; \beta \xrightarrow{SP} (\theta : \exists x B_1, \alpha); (\theta : \exists x B_2, \alpha) ; \beta \xrightarrow{SP^*} S/F
\]

But then we must have the computation
\[
(\theta : B_1[x := x'], \alpha); (\theta : B_2[x := x'], \alpha); \beta \xrightarrow{SP^*} S/F
\]

because \( (\theta : \exists x B_2, \alpha) \xrightarrow{SP^*} (\theta : B_2[x := x'], \alpha) \). So by the induction hypothesis, we have that
\[
(\theta : B[x := x'], \alpha); \beta \xrightarrow{SP^*} S/F
\]

So, adding one step onto the front of the computation,
\[
(\theta : \exists x B, \alpha); \beta \xrightarrow{SP^*} S/F
\]

This completes the proof of the converse direction.

\[ \square \]

Finally, we should note that the disjunctive unfolding of a formula is classically equivalent to the formula; this follows from the distributivity of \& and \( \exists \) over \( \lor \).

### 4.3 Axioms for Sequential Validity

With the notion of disjunctive unfolding in hand, we can now present the axioms SEQ describing validity relative to SP. Like the parallel axioms PAR, these axioms express how the signs \( S \) and \( F \) "distribute" over the goal formula connectives. In this section, I will prove these axioms sound, and in later sections I will prove the completeness results about LKE+SEQ which correspond to those of the last chapter about LKE+PAR.
## Chapter 4. Characterising Sequential Systems

### Success axioms:

<table>
<thead>
<tr>
<th>$S(\ldots)$</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: S(s = t) \rightarrow s = t$</td>
<td>$s = t \rightarrow S(s = t)$</td>
<td></td>
</tr>
<tr>
<td>$&amp;: S(B &amp; C) \rightarrow S(B) &amp; S(C)$</td>
<td>$S(B) &amp; S(C) \rightarrow S(B &amp; C)$</td>
<td></td>
</tr>
<tr>
<td>$\lor: S(B \lor C) \rightarrow S(B) \lor (F^Y(B) &amp; S(C))$</td>
<td>$S(B) \lor (F^Y(B) &amp; S(C)) \rightarrow S(B \lor C)$</td>
<td></td>
</tr>
<tr>
<td>$\exists: S(\exists x B) \rightarrow \exists x S(B)$</td>
<td>$\exists x S(B) \rightarrow S(\exists x B)$ (*a)</td>
<td></td>
</tr>
</tbody>
</table>

### Failure axioms:

<table>
<thead>
<tr>
<th>$F(\ldots)$</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: F(s = t) \rightarrow \neg s = t$</td>
<td>$\neg s = t \rightarrow F(s = t)$</td>
<td></td>
</tr>
<tr>
<td>$&amp;: F(B &amp; C) \rightarrow F(B) \lor (S(B) &amp; F(C))$</td>
<td>$(1) F(B) \rightarrow F(B &amp; C)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(2) F(B) \lor F(C) \rightarrow F(B &amp; C)$ (*a)</td>
</tr>
<tr>
<td>$\lor: F(B \lor C) \rightarrow F(B) &amp; F(C)$</td>
<td>$F(B) &amp; F(C) \rightarrow F(B \lor C)$</td>
<td></td>
</tr>
<tr>
<td>$\exists: F(\exists x B) \rightarrow \forall x F(B)$</td>
<td>$\forall x F^N(B) \rightarrow F(\exists x B)$</td>
<td></td>
</tr>
</tbody>
</table>

### Miscellaneous axioms:

<table>
<thead>
<tr>
<th>$F^N/F^Y$:</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F^N(A) \rightarrow F^Y(A)$</td>
<td>$F^N(P(t_1, \ldots, t_n)) \rightarrow$</td>
<td></td>
</tr>
</tbody>
</table>

### Side-conditions:

(*a) $B$ contains no predicate calls

(*b) $A'$ is a predicate unfolding of $A$, and $\sigma$ is either $S$ or $F^Y$

(*c) $A'$ is the disjunctive unfolding of $A$, and $\sigma$ is either $S$ or $F^Y$

Figure 4-3: SEQ axioms characterising sequential connectives. $F$ means either $F^Y$ or $F^N$, its use being consistent throughout each axiom.
Chapter 4. Characterising Sequential Systems

The axioms are in Figure 4-3. See Appendix A for examples of the use of LKE+SEQ in deriving sequents.

As with the LKE and PAR rules, we must prove them sound. In this chapter, I will use "validity" to mean "validity with respect to SP," unless stated otherwise.

**Theorem 4.3.1 (Validity of SEQ axioms)** Each instance of the axiom schemata in SEQ is a valid sequent.

**Proof.** One case for each arrow direction of each axiom. We will assume that \( \theta \) is a substitution which makes \( S \) closed.

\[ S(\equiv) : \text{As in the case for PAR. For } \theta \text{ which makes the sequent closed, } s\theta = t\theta \]
succeeds iff \( s \) and \( t \) are identical under \( \theta \), iff \( s\theta = t\theta \) is valid.

\[ S(\&): \text{As in the case for PAR. Since } B \text{ and } C \text{ are closed under } \theta, \text{ their conjunction succeeds iff each succeed independently.} \]

\[ S(V), \text{ right: If } S(B) V (F(B) & S(C)) \text{ is valid under } \theta, \text{ then either } B\theta \text{ succeeds or else } B\theta \text{ fails and } C\theta \text{ succeeds. But since } (()) : (B \lor C)\theta \stackrel{SF}{\rightarrow} ((()) : B\theta);((()) : C\theta), \]

\[ (B \lor C)\theta \text{ also succeeds.} \]

\[ S(V), \text{ left: } (()) : (B \lor C)\theta \stackrel{SF}{\rightarrow} ((()) : B\theta);((()) : C\theta) \stackrel{SP^*}{\rightarrow} (\theta' : \epsilon);\beta. \text{ So either } (()) : B\theta \text{ must succeed, or else } (()) : B\theta \text{ must fail and } (()) : C\theta \text{ succeed.} \]

\[ S(\exists), \text{ right: Assume that } \theta \text{ makes the sequent closed, and that the antecedent is valid; that is, that there is a } t \text{ such that } B[x := t]\theta \text{ succeeds. Since } B \text{ contains no predicate calls, } (\exists x B)\theta \text{ must either succeed or fail. If it were to fail, then (by Theorem 2.5.2) every } B[x := s]\theta \text{ would fail; so it must also succeed.} \]

\[ S(\exists), \text{ left: By the "succeeds-one-succeeds" theorem (Theorem 2.5.5), if } (\exists x B)\theta \text{ succeeds, then there must be some closed } t \text{ such that } B[x := t]\theta \text{ succeeds.} \]

For the \( F \) rules, we must prove validity for both \( F^V \) and \( F^N \).

\[ F(=): \text{As in the case for PAR. For } \theta \text{ which makes the sequent closed, } s\theta = t\theta \]
fails iff \( s \) and \( t \) are non-identical closed terms, iff \( \neg(s\theta = t\theta) \) is valid.
Chapter 4. Characterising Sequential Systems

\[ \text{F(\&), left: } (\alpha : (B\land C) \theta \xrightarrow{SP} (\alpha : B\theta, C\theta) \xrightarrow{SP*} \epsilon. \text{ So either } B\theta \text{ fails without even reaching } C\theta, \text{ or else } B\theta \text{ succeeds but } C\theta \text{ fails. If } (B\land C)\theta \text{ fails flatly, then so do the subsidiary computations. (This is clearly not all the information we can extract from the fact that } (B\land C)\theta \text{ fails; see the discussion on the inversion principle at the end of this chapter.)} \]

\[ \text{F(\&), right, 1: If } (\alpha : (B\theta) \xrightarrow{SP} \epsilon, \text{ we can graft the formula } C\theta \text{ onto every closure in that computation to get a failing computation of } (\alpha : B\theta, C\theta). \text{ But } (\alpha : (B\land C)\theta) \xrightarrow{SP} (\alpha : B\theta, C\theta), \text{ so } (B\land C)\theta \text{ fails too. If } B\theta \text{ fails flatly, then so does the new computation.} \]

\[ \text{F(\&), right, 2: If the antecedent is valid under } \theta, \text{ then either } B\theta \text{ fails or } C\theta \text{ fails. If } B\theta \text{ fails, then } (\alpha : B\theta, C\theta) \text{ must fail too; so } (B\land C)\theta \text{ fails. If } B\theta \text{ does not fail, then (since it contains no predicate calls) it must succeed, and } C\theta \text{ must fail for the antecedent to be valid; so again } (\alpha : B\theta, C\theta) \text{ must fail.} \]

\[ \text{F(\lor): As in the case for PAR. } (\alpha : B\theta); (\alpha : C\theta) \text{ fails iff both closures fail independently; and fails flatly iff both closures fail flatly.} \]

\[ \text{F(\exists), left: As in the case for PAR. By the "fails-all-fail" theorem (Theorem } 2.5.2), \text{ if } (\exists x B)\theta \text{ fails, then every instance of } B\theta \text{ fails, and the restriction to flat failure holds as well.} \]

\[ \text{F(\exists), right: As in the case for PAR. If every instance of } B\theta \text{ fails flatly, we can (by the techniques of this case in the PAR validity proof) prove that } (\exists x B)\theta \text{ fails.} \]

\[ \text{F}_N/F_Y: \text{ If } A \text{ fails flatly, then it clearly fails.} \]

\[ \text{Disj, Unf: by Theorems 3.3.2 and 4.2.2.} \]

\[ \text{F}_N(P): \text{ Clearly, no predicate call can fail without performing at least one Defined Predicate step.} \]

\[ \square \]

The reason for the restriction on the \( F(\&), \) right, 2 rule should be clear from the example given in the last section. The corresponding example justifying the restriction on the \( S(\exists), \) right rule is the following query:

\[ \exists x((x = 0 \& \text{Loop}()) \lor x = 1) \]
Obviously, the query \((1 = 0 & \text{Loop}(x)) \lor 1 = 1\) succeeds, so the assertion \(\exists x S((x = 0 & \text{Loop}(x)) \lor x = 1)\) is valid; but the query itself does not succeed. As in the case for \(F(&)\), though, the disjunctive unfolding of the query (in this case \(\exists x(x = 0 & \text{Loop}(x)) \lor \exists x(x = 1)\)) behaves properly.

Finally, we have the comprehensive Soundness result for LKE+SEQ.

**Theorem 4.3.2 (Soundness of LKE+SEQ)** Every sequent derivable in LKE+SEQ is valid.

**Proof.** By induction on the structure of the derivation. Cases are from the Validity theorems for LKE and SEQ.

\[\square\]

### 4.4 Completeness: Closed Assertions

We have the same kinds of completeness results for LKE+SEQ with respect to sequential validity as we did for LKE+PAR with respect to parallel validity. Many of the theorems are proved in very similar ways as those in the last chapter; I will omit these details, and concentrate on the cases in the proofs which are different from those for PAR.

The completeness theorem suggests a possible (inefficient) strategy for deciding sequents consisting of signed formulae. Informally, we decompose the formulae in the sequent using the appropriate connective axioms from SEQ. If we come to formulae which do not meet the side-conditions on the appropriate rules, we apply a finite number of predicate unfolding steps, followed possibly by a disjunctive unfolding step. We then proceed as before. This strategy will always work for valid sequents, because each step decreases the number of steps in the computation of the sequent.

**Theorem 4.4.1 (Closed Completeness, stage 1)** All valid sequents \(S\) which have only equalities in the antecedent, and only equalities and \(F^N\) formulae in the consequent, are derivable in LKE+SEQ.
Proof. As in the proof of the corresponding theorem for PAR, by induction on the number of connectives and equality formulae within $F^N$ signs. If this number is 0, then by the Completeness Theorem for Equality, Theorem 3.4.3, the result holds.

Otherwise, let $S$ be $\Gamma \rightarrow \Delta, F^N(D_1), \ldots, F^N(D_m)$, where the $\Delta$ formulae are all equalities. Cases are on $D_1$. We will derive sequents which must also be valid, and which have fewer connectives and equality formulae within $F^N$ signs.

$D_1 \equiv (s = t)$: as in the proof for PAR.

$D_1 \equiv B \& C$: Assume that $S$ is valid. There are three subcases. If $B$ contains no predicate calls, then under every substitution, $B$ must either succeed or fail. Consider the sequent $S'$ formed by replacing $F^N(B \& C)$ by $F^N(B), F^N(C)$. If $S'$ were invalid, then there would be a substitution under which neither $F^N(B), F^N(C)$ nor any of the other signed formulae in $S$ would be valid; but if that were true, $B\theta$ would succeed (since it cannot diverge or fail non-flatly), and $C\theta$ would not fail flatly, making $F^N(B \& C)\theta$ invalid, and thus making $S$ invalid. So $S'$ must be valid; by the induction hypothesis, it is derivable; and $S$ can be derived from it by $V, r$ and Cut with $F(\&), right, 2$ from SEQ.

Finally, if $B$ contains predicate calls and its key subformula is a disjunction $B_1 \lor B_2$, then the disjunctive unfolding of $B \& C$ must be $(B^{B_1} \& C) \lor (B^{B_2} \& C)$. Let $S_1$ be $S$ with $F^N(B \& C)$ replaced by $F^N(B^{B_1} \& C)$, and let $S_2$ be $S$ with $F^N(B \& C)$ replaced by $F^N(B^{B_2} \& C)$. Then $S_1$ must be valid, because if there were no flatly failing computation for $B^{B_2} \& C$ then there could be no flatly failing
computation for B&C, since the start of the computation of B&C is essentially the same except for the v step. Similarly, S₂ must be valid because otherwise the tail of the computation of B&C could not result in flat failure. But since both S₁ and S₂ have one fewer connective (the v is missing), by the induction hypothesis they must be both derivable. S is derivable from S₁ and S₂ by an application of &., r, an application of Cut with F(v), right, and an application of Cut with Disj, right from SEQ.

\[ \mathcal{D}_1 \equiv B \lor C : \text{If } B \lor C \text{ fails flatly under } \theta, \text{ then both } B \text{ and } C \text{ must fail flatly under } \theta. \text{ So both } S \text{ with } B \lor C \text{ replaced by } B \text{ and } S \text{ with } B \lor C \text{ replaced by } C \text{ must be valid; by the induction hypothesis, they are both derivable; and } S \text{ can be derived from them by using the } \&., r \text{ rule and Cut with } F(\lor), \text{ right from SEQ.} \]

\[ \mathcal{D}_1 \equiv \exists x B: \text{ As in the proof for PAR: we can use } \lor, r \text{ and } F(\exists), \text{right from SEQ to derive } S \text{ from a valid sequent of lower measure.} \]

The results in the last chapter about predicate unfoldings and about failing computations both extend to computations in SP. They are not simple consequences of those results, but the proofs are similar.

Lemma 4.4.2 If a backtrack stack \( \beta \) fails, then some predicate unfolding of \( \beta \) can fail flatly.

Proof. As in the last chapter.

\[ \square \]

Theorem 4.4.3 (Closed Completeness, stage 2a) All valid sequents \( S \) of the form \( [\rightarrow F^y(A)] \), where \( A \) is a closed formula, are derivable in LKE+SEQ.

Proof. As in the last chapter. We can unfold \( A \) until it fails flatly.

\[ \square \]

Theorem 4.4.4 (Closed Completeness, stage 2b) All valid sequents \( S \) of the form \( [\rightarrow S(A)] \), where \( A \) is a closed formula, are derivable in LKE+SEQ.
Proof. By the definition of validity, \((() : A)\) must succeed. We can proceed by induction on the length of this computation. Cases are on the form of \(A\).

\(A \equiv (s = t)\): As in the theorem for PAR. \(s\) and \(t\) are closed, and so must be identical for them to unify. We can therefore derive \(S\) from an Eq axiom and an application of Cut with the \(S(=)\), right axiom of SEQ.

\(A \equiv B \& C\): As in the theorem for PAR. \(B\) and \(C\) must succeed independently, one after another, each with a shorter computation than \(B \& C\). \(S\) must therefore be derivable from \([\rightarrow S(B)]\) and \([\rightarrow S(C)]\) by \& \(r\) and Cut with \(S(\&)\), right of SEQ.

\(A \equiv B V C\): The first step of the computation is \((() : B V C) \rightarrow (() : B); (() : C)\). Clearly either \(B\) succeeds (in which case \(S\) is derivable from \([\rightarrow S(B)]\) by Thin \(r\) and \(V \& r\) and Cut with \(S(V)\), right from SEQ), or else \(B\) fails and \(C\) succeeds (in which case \(S\) is derivable from \([\rightarrow F(B)]\) and \([\rightarrow S(C)]\) by \& \(r\), Thin \(r\), and \(V \& r\) and Cut with \(S(V)\), right from SEQ).

\(A \equiv \exists x \ B\): There are three subcases. First, let \(B\) have no predicate calls. By the "succeeds-one-succeeds" theorem (Theorem 2.5.5), there must be a \(t\) such that \(B[x := t]\) succeeds. This successful computation is the same length or shorter than the computation of \(B\), namely one step shorter than the computation of \(\exists x \ B\); so \([\rightarrow S(B[x := t])]\) is derivable, and \(S\) can be derived from it by \(\exists \, r\) and Cut with \(S(\exists)\), right from SEQ.

If \(B\) contains predicate calls, and its key subformula is a predicate call, then that key subformula must come to the top of the goal stack during the course of the successful computation. The predicate 1-expansion \(A'\) of \(A\) in which the key subformula is expanded therefore must take one fewer step than that of \(A\). \(S\) is therefore derivable from \([\rightarrow S(A')]\) by Cut with Unf \(r\) of SEQ.

Finally, if \(B\) contains predicate calls but its key subformula is a disjunction \(B_1 \lor B_2\), then by Theorem 4.2.2, \(A\)'s disjunctive unfolding \(A^{B_1} \lor A^{B_2}\) must succeed. We can follow similar reasoning to the \(\lor\) case, above, to show that either \([\rightarrow S(A^{B_1})]\) is valid, or both \([\rightarrow F(A^{B_1})]\) and \([\rightarrow S(A^{B_1})]\) are valid; but all of these will correspond to shorter computations than \(A\), and so are derivable; so \(S\) is
derivable by using $\land$, $\lor$ and $\forall$, $\exists$ from LKE, Cut with $S(\forall)$, right from SEQ, and Cut with $\text{Disj}, r$ of SEQ.

$A \equiv P(t_1, \ldots, t_n)$: As in the theorem for PAR; as in the last case, we can use Cut with Unf, right from SEQ to get a sequent which is valid and takes fewer computation steps, and which thus is derivable.

The rest of the stages are as in the last chapter.

**Theorem 4.4.5 (Closed Completeness, stage 3)** All valid sequents $S$ which have only equalities in the antecedent, and only equalities and signed formulae in the consequent, and where no free variable appears in any $S$ or $F^V$ assertion, are derivable in LKE+SEQ.

**Proof.** As in the last chapter. We can either thin out all but a valid $S$ or $F^V$ formula, or thin out all the $S$ and $F^V$ formulae, and by stages 1, 2a, and 2b, we will be able to derive the resulting sequent.

Theorem 4.4.6 (Closed Completeness, stage 4) All valid sequents $S$ in which no free variable appears in an $S$ or $F^V$ subassertion, and no signed formula appears in a negative context, are derivable in LKE+SEQ.

**Proof.** As in the last chapter: by induction on the total number of connectives outside signed formulae.

Finally, we have the important results about the characterisation of successful and failing queries corresponding to that in the last chapter.

**Theorem 4.4.7 (Characterisation of SP)** A goal formula $A$ succeeds in SP iff the sequent $[\rightarrow S(\exists[A])]$ is derivable in LKE+SEQ; it fails in SP iff the sequent $[\rightarrow F^V(\exists[A])]$ is derivable in LKE+SEQ.
Proof. As in the last chapter: from the Soundness and Completeness theorems.

So just as LKE+PAR characterised the two outer circles from the Venn diagram (Figure 2-6), LKE+SEQ has succeeded in characterising the two innermost circles from that diagram. A query A is in the innermost failure set iff \([\rightarrow F(\exists[A]])\) is derivable; a query A is in the innermost success set iff \([\rightarrow S(\exists[A]])\) is derivable. Because of the completeness property of SOS/sa, we have also characterised the queries which fail in SOS/sa.

4.5 Completeness: Predicate-Free Assertions

The results about predicate-free assertions in PAR extend to SEQ in a simpler way than did the results about closed assertions. This is because predicate-free backtrack stacks either succeed or fail, and do not diverge; so SP is actually complete with respect to SOS for such backtrack stacks.

Theorem 4.5.1 (Flat Completeness, stage 1) All valid sequents S containing no signed formulae are derivable in LKE.

Proof. If S is valid with respect to SP, then it is also valid with respect to SOS; thus, by the Flat Completeness, stage 1 theorem in the last chapter, it is derivable.

Theorem 4.5.2 (Flat Completeness, stage 2) All valid sequents S containing no predicate calls are derivable in LKE+SEQ.

Proof. As in the last chapter. The more restrictive side-conditions do not apply, since none of the signed formulae contain predicate calls; the different forms
of some of the rules do not seriously affect the proof.

\[\square\]

The results about hierarchical programs extend to SEQ as well.

**Theorem 4.5.3** If \( \Pi \) is hierarchical, then all valid sequents \( S \) are derivable in LKE+SEQ.

**Proof.** As in the last chapter: we can unfold predicate calls until none are left in the sequent.

\[\square\]

### 4.6 Success for SOS/so

The last circle on the Venn diagram in Figure 2–6 left to be characterised is the middle success set – the set of queries succeeding in SOS/so. The set of *failing* queries for SOS/so is exactly that for SOS, so we have no need of further characterising that set; and for the successful queries, it turns out that a proof system essentially consisting of the success axioms from SEQ and the failure axioms from PAR will serve to characterise this class.

As we saw in Chapter 2, the set of successful queries for SOS/so is smaller than the set of successful queries for SOS because the sequential “or” misses some solutions; but it is also bigger than the set of successful queries for SP, because the parallel “and” causes more queries to fail, and this has the knock-on effect of allowing some more queries to succeed.

This effect of the expansion of the set of failing queries is illustrated well in the set of axioms PASO (“Parallel And, Sequential Or”; Fig. 4–4). The Success and Miscellaneous axioms of PASO are exactly those from SEQ, but the Failure axioms are exactly those from PAR. We are therefore able to prove more things to be failing than in SEQ (for example, \( \text{Loop()} \& \text{false} \)); and because the \( S(V) \) axioms depend on failure, we are therefore able to prove more things successful than in
### Chapter 4. Characterising Sequential Systems

#### Success axioms:

<table>
<thead>
<tr>
<th>$S(\ldots)$</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: \ S(s = t) \rightarrow s = t$</td>
<td>$s = t \rightarrow S(s = t)$</td>
<td></td>
</tr>
<tr>
<td>$&amp;: \ S(B &amp; C) \rightarrow S(B) &amp; S(C)$</td>
<td>$S(B) &amp; S(C) \rightarrow S(B &amp; C)$</td>
<td></td>
</tr>
<tr>
<td>$\lor: \ S(B \lor C) \rightarrow S(B) \lor (F^Y(B) &amp; S(C))$</td>
<td>$S(B) \lor (F^Y(B) &amp; S(C)) \rightarrow S(B \lor C)$</td>
<td></td>
</tr>
<tr>
<td>$\exists: \ S(\exists x \ B) \rightarrow \exists x \ S(B)$</td>
<td>$\exists x \ S(B) \rightarrow S(\exists x \ B)$ ($*a$)</td>
<td></td>
</tr>
</tbody>
</table>

#### Failure axioms:

<table>
<thead>
<tr>
<th>$F(\ldots)$</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=: \ F(s = t) \rightarrow \neg s = t$</td>
<td>$\neg s = t \rightarrow F(s = t)$</td>
<td></td>
</tr>
<tr>
<td>$&amp;: \ F(B &amp; C) \rightarrow F(B) \lor F(C)$</td>
<td>$F(B) \lor F(C) \rightarrow F(B &amp; C)$</td>
<td></td>
</tr>
<tr>
<td>$\lor: \ F(B \lor C) \rightarrow F(B) &amp; F(C)$</td>
<td>$F(B) &amp; F(C) \rightarrow F(B \lor C)$</td>
<td></td>
</tr>
<tr>
<td>$\exists: \ F(\exists x \ B) \rightarrow \forall x \ F(B)$</td>
<td>$\forall x \ F^N(B) \rightarrow F(\exists x \ B)$</td>
<td></td>
</tr>
</tbody>
</table>

#### Miscellaneous axioms:

<table>
<thead>
<tr>
<th>$F^N / F^Y$</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F^N / F^Y$:</td>
<td>$F^N(A) \rightarrow F^Y(A)$</td>
<td></td>
</tr>
<tr>
<td>Unf: $\sigma(A) \rightarrow \sigma(A')$ ($*b$)</td>
<td>$\sigma(A') \rightarrow \sigma(A)$ ($*b$)</td>
<td></td>
</tr>
<tr>
<td>Disj: $\sigma(A) \rightarrow \sigma(A')$ ($*c$)</td>
<td>$\sigma(A') \rightarrow \sigma(A)$ ($*c$)</td>
<td></td>
</tr>
<tr>
<td>$F^N(\mathbb{P})$:</td>
<td>$F^N(\mathbb{P}(t_1, \ldots , t_n)) \rightarrow$</td>
<td></td>
</tr>
</tbody>
</table>

#### Side-conditions:

- ($*a$) $B$ contains no predicate calls
- ($*b$) $A'$ is a predicate unfolding of $A$, and $\sigma$ is either $S$ or $F^Y$
- ($*c$) $A'$ is the disjunctive unfolding of $A$, and $\sigma$ is either $S$ or $F^Y$

Figure 4-4: PASO axioms characterising SOS/so. $F$ means either $F^Y$ or $F^N$, its use being consistent throughout each axiom.
SEQ (for example, \((\text{Loop}() \& \text{false}) \lor \text{true}\), the example from Chapter 2). See the Appendix for a derivation of the success of this query in LKE+PASO.

We must prove these axioms to be sound, and prove the equivalents of the Closed Completeness results, of course. However, many of the results we need to prove follow directly from those in this chapter and the previous one, and others have very similar proofs. I will briefly outline the results, noting only where they differ from those for PAR and SEQ.

**Theorem 4.6.1 (SOS/so Equivalence of Disjunctive Unfoldings)**

If \( A \xrightarrow{\text{Disj}} A' \), then \((\theta : A, \alpha); \beta \) succeeds (fails) in SOS/so iff \((\theta : A', \alpha); \beta \) succeeds (fails) in SOS/so.

**Proof.** This is much the same as in the case for SEQ; I will sketch it here. The proof is by induction on the depth of the key subformula, a disjunction, in \( A \).

Case depth = 0: \( A \) is itself a disjunction, so its disjunctive unfolding is itself. The result holds trivially.

Case depth > 0: \( A \) can be either a conjunction or an existential formula.

- \( A \equiv B \& C \), and the key subformula, \( B_1 \lor B_2 \), is in \( B \). If one of the following goal stacks succeeds (resp. fails) in SOS/so, all the rest succeed (resp. fail) in SOS/so:

  \[
  (\theta : B \& C, \alpha) \\
  (\theta : B, \alpha) \\
  (\theta : B_1 \lor B_2, C, \alpha) \\ *
  (\theta : B_1, C, \alpha); (\theta : B_2, C, \alpha) \\
  (\theta : B_1 \& C, \alpha); (\theta : B_2 \& C, \alpha) \\
  (\theta : (B_1 \& C) \lor (B_2 \& C), \alpha)
  \]

  where the step * is from the induction hypothesis. Since \( A' \equiv (B_1 \& C) \lor (B_2 \& C) \), the result holds.
\( A \equiv B \land C, \) and the key subformula, \( C_1 \lor C_2, \) is in \( C. \) If one of the following goal stacks succeeds (resp. fails) in SOS/so, all the rest succeed (resp. fail) in SOS/so:

\[
(\theta : B \land C, \alpha)
\]

\[
(\theta : B, C, \alpha)
\]

\[
(\theta : B, C^1 \lor C^2, \alpha) \ast
\]

\[
(\theta : B, C^1, \alpha); (\theta : B, C^2, \alpha)
\]

\[
(\theta : B \land C^1, \alpha); (\theta : B \land C^2, \alpha)
\]

\[
(\theta : (B \land C^1) \lor (B \land C^2), \alpha)
\]

where the step \( \ast \) is from the induction hypothesis. Since \( A' \equiv (B \land C^1) \lor (B \land C^2), \) the result holds.

\( A \equiv \exists x B. \) If one of the following goal stacks succeeds (resp. fails) in SOS/so, all the rest succeed (resp. fail) in SOS/so:

\[
(\theta : \exists x B, \alpha)
\]

\[
(\theta : B[x := x'], \alpha)
\]

\[
(\theta : B^1[x := x'] \lor B^2[x := x'], \alpha) \ast
\]

\[
(\theta : B^1[x := x'], \alpha); (\theta : B^2[x := x'], \alpha)
\]

\[
(\theta : B^1[x := x'], \alpha); (\theta : B^2[x := x''], \alpha)
\]

\[
(\theta : \exists x B^1, \alpha); (\theta : B^2[x := x''], \alpha)
\]

\[
(\theta : \exists x B^1, \alpha); (\theta : \exists x B^2, \alpha)
\]

\[
(\theta : \exists x B^1 \lor \exists x B^2, \alpha)
\]

where the step \( \ast \) is from the induction hypothesis. Since \( A' \equiv \exists x B^1 \lor \exists x B^2, \) the result holds.
Theorem 4.6.2 (Validity of PASO Axioms) Each instance of the axiom schemata in PASO is valid with respect to SOS/so.

Proof. The Failure group is identical to the one found in PAR. The only signed formulae they contain are $F$-signed formulae, which are valid wrt SOS/so iff they are valid wrt SOS (by the completeness results in Chapter 2); so, since we have proven them valid wrt SOS in the validity proof for PAR, they are also valid wrt SOS/so. The same holds for the Miscellaneous axioms that have only $F$-signed formulae.

The proofs for the other axioms are much as in the case for SEQ. The “and” is now parallel, but the logic remains the same: if $(B\theta \& C\theta)$ is closed, then it succeeds iff both $B\theta$ and $C\theta$ succeed independently. The validity of the Disj axioms follows from the operational equivalence of disjunctive unfolding for SOS/so (Theorem 4.6.1). All other axioms follow the pattern of SEQ.

In the completeness results, the only stage that needs to be re-proved is Closed Completeness, stage 2b:

Theorem 4.6.3 (Closed Completeness, stage 2b) All sequents $S$ valid wrt SOS/so and of the form $[-p \rightarrow S(A)]$, where $A$ is a closed formula, are derivable in LKE+PASO.

Proof. Almost identical to the case for SEQ. The only case that is significantly different is $S(\exists)$. There, we must prove that if $\exists x B$ is closed and succeeds, then $[-\rightarrow S(\exists x B)]$ is derivable, given the induction hypothesis. There are three subcases.

In the case that $B$ contains no predicate calls, we can apply the $S(\exists)$, right axiom straightforwardly, because the “succeeds-one-succeeds” theorem (Theorem 2.5.5) guarantees that there is a witness.
If \( B \) contains predicate calls and its key subformula is a predicate call, then the proof is not quite so straightforward as in the case for SEQ. Because we have parallel "and", the closure may be split up into several due to some disjunction to the right of the key subformula; for example,

\[
(\theta_1 : \ldots \cdot P(\ldots), \ldots, B \lor C) \xrightarrow{\text{SOS/so}} (\theta_1 : \ldots \cdot P(\ldots), \ldots, B); (\theta_1 : \ldots \cdot P(\ldots), \ldots, C)
\]

However, it is still the case that since the query succeeds, the key subformula must eventually be computed in at least one of the resultant closures. Thus, the predicate 1-unfolding of \( B \) in which the key subformula is unfolded must have an SOS/so-computation of fewer steps. The induction hypothesis applies, and we can therefore derive \( \rightarrow S(\exists x \ B) \) from the derivable \( \rightarrow S(\exists x \ B') \), where \( B' \) is this predicate 1-unfolding of \( B \).

If \( B \) contains predicate calls and its key subformula is a disjunction, then (by the operational equivalence of disjunctive unfoldings, Theorem 4.6.1) the disjunctive unfolding of \( \exists x \ B, \exists x B_1 \lor \exists x B_2 \), must also succeed. But then, as in the case for SEQ, we can use the \( S(V) \), right axiom to derive \( \rightarrow S(\exists x B_1 \lor \exists x B_2) \) from either \( \rightarrow S(\exists x B_1) \) or from \( \rightarrow F(\exists x B_1) \) and \( \rightarrow S(\exists x B_2) \).

\( \square \)

Stages 1 and 2a of Closed Completeness are about failing formulae, and thus follow from the corresponding stages from PAR. Stages 3 and 4 follow the proofs for PAR and SEQ exactly. We therefore have the following important characterisation result corresponding to those for PAR and SEQ:

**Theorem 4.6.4 (Characterisation of SOS/so)** A goal formula \( A \) succeeds in SOS/so iff the sequent \( \rightarrow S(\exists [A]) \) is derivable in LKE+PASO; it fails in SOS/so iff the sequent \( \rightarrow F^V(\exists [A]) \) is derivable in LKE+PASO.

**Proof.** As in the case for PAR and SEQ: from the Soundness and Completeness theorems.

\( \square \)

Finally, as with SEQ, the Flat Completeness results go through unchanged because all sequents containing no predicate calls have the property that they
are either valid with respect to all variants of SOS, or invalid with respect to all variants.

These soundness and completeness results complete our logical characterisation of the queries succeeding and failing in the various control disciplines. The characterisation is especially pleasing because the three characterising proof systems share the common rules LKE, and because the PASO rules integrate the PAR and SEQ rules in a simple and intuitively clear way.

4.7 Discussion

Some comments from the Discussion section of Chapter 3 apply to the material in this chapter as well. These include the discussion of more practical failure rules. Here, I concentrate on issues surrounding SEQ in particular, and its correspondence to SP; these comments apply equally to PASO and its correspondence to SOS/so.

4.7.1 SEQ as a Characterisation of SP

Our goal, in this chapter, has been to find a logical characterisation of sequential logic programming. Is LKE+SEQ such a characterisation? By Theorem 4.4.7, it is a characterisation, but is it logical? The answer to this question hinges on whether we consider the non-compositional elements of SEQ (the predicate and disjunctive unfolding rules) to be "logical" enough.

Shoenfield [Sho67] has discussed the related "characterisation problem" for formal systems: the problem of finding "a necessary and sufficient condition that a formula of [formal system] $F$ be a theorem of $F$." He writes:

There is a trivial solution to the characterization problem for a theory $T$: a formula is a theorem iff it has a proof. This is an unsatisfactory solution because the condition for $A$ to be a theorem depends upon all
formulas which might appear in a proof of $A$. In a satisfactory solution, the condition must depend only upon $A$ and formulas closely related to $A$.

An adaptation of Shoenfield's criterion to the characterisation of logic programming systems might be that the condition for a query $A$ to succeed (fail) must depend only upon the conditions for success (failure) of $A$ and formulae closely related to $A$.

Does SEQ meet this criterion? The compositional axioms certainly do, but what about the unfolding axioms? In support of the claim that they do is the fact that the unfoldings of $A$ are classically equivalent to $A$, given the definitions in the program $H$. The unfoldings of $A$ also preserve much of the structure of $A$, changing only single predicate application subformulae or duplicating most of the formula. In this sense, we can say that the unfoldings of $A$ are "closely related" to $A$.

On the other hand, they are not as closely related to $A$ as we would expect given other logical systems. In traditional systems, the meaning of a formula is dependent only on the meanings of its immediate subformulae, possibly (in the case of the quantifiers) with a substitution applied to them. Whether we can come up with logical characterisations of sequential logic programming that more closely approximate this ideal is an open problem.

A related question is the following: do the operational and denotational semantics of sequential logic programming given in the past meet Shoenfield's criterion? These semantics are certainly "compositional" in the sense used to discuss denotational semantics: the denotation of a formula is a function of the denotations of its immediate subformulae. However, the success or failure of a query depends not on that of its subformulae, but on the properties of its denotation, which can be computed only by a process similar to computations in the operational semantics. It is difficult to say, therefore, that these semantics truly meet Shoenfield's criterion.
Of particular importance here is the reification of unification in denotational semantics of Prolog. Unification is an important algorithm which logic programmers must understand to know how their programs work. But it is used in the resolution method to solve a particular, important problem—the problem of finding witnesses for quantified variables. The primacy of the logical semantics of the existential quantifier remains.

It is this logical semantics—that the quantified formula is true (false) iff there is (is not) a witness—that is captured better by SEQ.

This is not to say that there are no other criteria for judging semantics, on which denotational semantics may be judged superior. Compositional, fully abstract denotational semantics have been extensively studied and are well understood. In some denotational semantics of Prolog, the cut and negation-as-failure operations are able to be described; in the formulation of logic programming that I use here, cut is not even definable and negation is not considered.

Whether denotational semantics will lead us to a logical characterisation of cut and negation as failure within the context of sequential logic programming remains to be seen, however. Future directions for the work in this chapter include an account of some form of negation and an if-then-else construct; see Chapter 6 for more details.

4.7.2 The Inversion Principle for SEQ

The Inversion Principle for natural deduction systems [Pra65] can be phrased as follows: the elimination rules (in our context, the antecedent rules) for a connective should allow us to extract all of the information about a formula A that would be necessary to allow us to conclude that A was true. This is not the case for the left-hand rules for $S(\exists)$ and $F(\&)$.

In the case of $S(\exists)$, for instance, we can conclude from the information that $\exists x B$ succeeds that there is a $t$ such that $B[x := t]$ succeeds. However, this information is not sufficient to prove that $\exists x B$ succeeds. This is one of the effects of having the non-compositional rules of predicate and disjunctive unfolding: the
completeness of the consequent rules depends on being able to transform a formula into one for which certain side-conditions are met. In this case, we can regain the inversion property by making the same restriction on the formula as holds in the introduction rule: that $B$ contain no predicate calls. If $B$ does contain predicate calls, then we must use the antecedent unfolding rule, just as we have to use the consequent unfolding rule when the formula appears in the consequent.

The case of $F(\&)$ is more problematic. We cannot make the restriction that $B$ contain no predicate calls, because it may be that $B$ contains predicate calls but fails before it makes any. One possibility for regaining an inversion property is to split the left-hand axiom into two:

$$F(B\&C) \rightarrow F^N(B) \lor F(A')$$

where $B$ contains predicate calls, and $A'$ is a predicate unfolding of $(B\&C)$; and

$$F(B\&C) \rightarrow F(B) \lor (S(B)\&F(C))$$

where $B$ contains no predicate calls. These rules are slightly clumsy, but do extract as much information out of antecedent signed formulae as possible.

In short, we can modify some of the rules in SEQ to regain an inversion property. Since this involves making changes and restrictions to the rules inessential to the proofs of soundness and completeness, I chose not to make these restrictions in SEQ for simplicity's sake.
Chapter 5

Approaches to Incompleteness

Although we have proven some useful completeness theorems about the proof systems in the last two chapters, we have not been able to prove absolute completeness: that every valid sequent is derivable. Because of some formal incompleteness results, we will never be able to prove such a completeness theorem, for any finitary proof system; but there are several ways in which we can, at least partially, escape the effect of these incompleteness results. In this chapter, I present the incompleteness theorems and some of the partial solutions.

There are two main incompleteness results, as discussed in the first section below. The first says that we will never be able to derive all valid closed sequents which have signed formulae in negative contexts, and follows from the non-existence of a solution to the Halting Problem. (We can deal with many of the important cases of this result by adding extra rules which I will describe.) The second result says that we will never be able to derive all valid sequents with free variables, even if they have no signed formulae in negative contexts, and is a version of Gödel’s Incompleteness Theorem.

The “mathematical” solution to these problems is to bring the proof theory closer to a kind of model theory, by allowing infinitary elements into the proof systems. Though these are not adequate solutions for practical theorem proving, they are useful in that they shed light on the extent to which the proof systems in question are complete. I discuss these methods in the second section of this chapter.
The more practical solution to some of the incompleteness problems is to add some form of induction. Many of the useful sequents with free variables which we cannot prove in the finitary systems in the last two chapters, can be proven if we add induction rules to the systems. Some of the variations on this approach are described in the last section.

5.1 Incompleteness

In the Closed Completeness results in the last two chapters, there were two restrictions on the class of valid sequents being proven derivable. The first was that signed formulae were barred from appearing in negative contexts; the second was that free variables were barred from appearing within signed formulae.

This is not to say that no valid sequent not appearing in the classes mentioned is derivable; many are, as for instance the Flat Completeness results show. So we can relax these restrictions to some extent, and show that wider classes of valid sequents are derivable. But we cannot remove either of them completely, even while maintaining the other. If we remove the first restriction, we will not be able to find a finitary proof system which is complete for that class of sequents, due to the unsolvability of the Halting Problem. This will also happen if we remove the second restriction, due this time to a version of Gödel's Incompleteness Theorem.

This section proves these incompleteness results, and points out a small relaxation of the first condition which will allow us to prove the derivability of another fairly useful class of valid sequents.

5.1.1 The Halting Problem and Divergence

If we maintain the restriction to closed sequents, can we relax the restriction to signed formulae only in positive contexts? The answer is no: although a finitary proof system (a proof system in which all judgments and derivations are of finite
size) can be complete for closed sequents of the form \([\rightarrow S(A)]\) and \([\rightarrow F(A)]\), it cannot be complete for closed sequents in general.

**Halting Problem Incompleteness**

**Theorem 5.1.1 (Halting Problem Incompleteness)** There is no sound, finitary proof system for sequents (as defined in this thesis) which has as theorems all valid, closed sequents of the forms \([S(A) \rightarrow] \) and \([F^Y(A) \rightarrow] \); and this is the case for validity with respect to any variant of SOS.

**Proof.** The theorems of a finitary proof system can be recursively enumerated. If there were a sound, finitary proof system with the stated completeness property, then we would be able to effectively decide whether a given closed query \(A\) terminates or not. The procedure is as follows: dovetail the computations for enumerating the theorems of the forms \([S(B) \rightarrow] \) and \([F^Y(B) \rightarrow] \), and for enumerating the theorems of LKE+PAR (or LKE+SEQ or LKE+PASO) of the forms \([\rightarrow S(B)] \) and \([\rightarrow F^Y(B)] \). If \(A\) succeeds as a query, \([\rightarrow S(A)] \) will be reached eventually; if \(A\) fails, \([\rightarrow F^Y(A)] \) will be reached; and if \(A\) diverges, both \([S(A) \rightarrow] \) and \([F^Y(A) \rightarrow] \) will be reached. Stop when one of these situations arises.

Since every variant of SOS is Turing-complete, this procedure would constitute a solution to the halting problem. Therefore there can be no such proof system.

\[\square\]

**Guaranteed Termination**

The Halting Problem Incompleteness theorem seems a little “unfair” in some ways. Some sequents are valid for the trivial reason that some instance of some signed formula in a negative context diverges. It seems that many of the cases in which we will be unable to prove a valid sequent happen when the sequent is of this form. This is unfair because in practice, we will seldom want to prove such sequents; we are more interested in proving properties of programs given known success or failure properties of predicates.
One important class of such sequents has assumptions of the form $S(A)$ or $F(A)$, in which each instance of $A$ either succeeds or fails. For instance, we may want to prove the sequent

$$S(\text{Even}(x)) \rightarrow S(\text{N}(x))$$

where $N$ tests whether its argument is a Peano integer, and $\text{Even}$ tests whether it is an even Peano integer. Every closed instance of this sequent is derivable in LKE+PAR or LKE+SEQ, even though the Closed Completeness theorem does not include it. In general, if we want to prove properties of predicates based on this kind of "type" or "structural" properties of their arguments, we will get sequents of this form. We would therefore like to prove a completeness result which takes in this class of sequents as well.

**Definition 5.1.2** A goal formula $A$ is guaranteed terminating (in a variant of SOS) if every instance of $A$ either succeeds or fails. We similarly say that a signed formula is guaranteed terminating if its formula is guaranteed terminating.

To facilitate the proof of completeness for guaranteed termination, I will introduce two simple rules which are sound with respect to any operational semantics, but were not needed for proving soundness or completeness results so far. The completeness of guaranteed termination could be proven without them, but not without repeating many of the cases in the other completeness proofs. Let the set of rules $GT$ be the following:

$$GT,S: \quad \frac{\Gamma \rightarrow F^Y(A), \Delta}{\Gamma, S(A) \rightarrow \Delta} \quad GT,F: \quad \frac{\Gamma \rightarrow S(A), \Delta}{\Gamma, F(A) \rightarrow \Delta}$$

where $F$ in $GT,F$ means either $F^Y$ or $F^N$.

**Theorem 5.1.3** The $GT$ rules are sound with respect to any variant of SOS; that is, if the premisses of an application of one of the GT rules are valid wrt the variant, then the conclusion is valid wrt the variant.

**Proof.** Consider $GT,S$. The premiss says that if all the $\Gamma$ assertions are valid under $\theta$, then either $A$ fails (in which case $S(A) \theta$ is invalid and the conclusion
is valid) or else one of the $\Delta$ assertions is valid under $\theta$ (in which case, again, the conclusion is valid).

The proof for $GT,F$ is similar; the reasoning is the same for both the $F^Y$ and $F^N$ cases.

The kind of completeness that these rules give us is the following consequence of Closed Completeness, stage 3.

**Theorem 5.1.4 (Completeness for Guaranteed Termination)**

All sequents $S$ valid wrt SOS (resp. SP, SOS/so) in which all assertions in the antecedent are equality formulae or guaranteed-terminating signed formulae, all assertions in the consequent are equality formulae or signed formulae, and no free variable appears within an $S$ or $F^Y$ sign, are derivable in LKE+PAR+GT (resp. LKE+SEQ+GT, LKE+PASO+GT).

**Proof.** By induction on the number of signed formulae in the antecedent. When this is not zero, consider the first such formula, $\sigma(A)$. $A$ either succeeds or fails; therefore (by Closed Completeness, stage 3) we can prove the sequent $[-p S(A),F^1'(A)]$.

If $\sigma$ is $S$, let $S'$ be $S$ with $S(A)$ removed from the antecedent and $F^Y(A)$ added to the consequent. $S'$ is derivable from $S$ using Cut and $[-S(A),F^Y(A)]$, so it is valid; by the induction hypothesis, it must be derivable; and $S$ is derivable from it by an application of the $GT,S$ rule.

The cases where $\sigma$ is $F^Y$ or $F^N$ are similar.

There is a further result corresponding to stage 4 of Closed Completeness.

**Theorem 5.1.5** All sequents $S$ valid wrt SOS (resp. SP, SOS/so) in which no free variable appears in an $S$ or $F^Y$ subassertion, and no signed formula appears in a negative context except guaranteed-terminating signed formulae, are derivable in LKE+PAR+GT (resp. LKE+SEQ+GT, LKE+PASO+GT).
Proof. As in the previous two chapters.

With the guaranteed-termination rules, we therefore have the ability to derive some important valid sequents. We cannot yet derive \( S(\text{Even}(x)) \rightarrow S(\text{N}(x)) \), since this contains free variables, but we can derive every instance of it.

Of course, with guaranteed termination, we still have the problem of actually guaranteeing the termination! We need to prove, essentially, that \([- S(A), F(A)]\) is valid, for each \( \sigma(A) \) appearing in a negative context. If such a sequent is closed or has no recursive predicates, then (by the completeness results) we can give a derivation for it. Otherwise, we may need to prove its validity by hand – or use the induction rules given in the last section of this chapter to derive it.

5.1.2 Gödel Incompleteness

In this section, I give a sketch of a proof that there is no sound, complete, finitary proof system for sequents without the restriction on free variables, even retaining the restriction against having signed formulae in a negative context. This proof is a variant of Gödel's proof of the incompleteness of arithmetic [Göd62]. For simplicity, the exact version of the theorem I will prove will be about different kinds of judgements than sequents. I trust that this will convince readers of the truth of the more specific theorem. The theorem applies to validity with respect to any variant of SOS.

Definition 5.1.6 A program-formula judgement in a language \( \mathcal{L} \) is an expression of the form \( \langle \Pi, \sigma(A) \rangle \), where \( \Pi \) is (some syntactic representation of) a program in \( \mathcal{L} \), and \( \sigma(A) \) is a signed formula of \( \mathcal{L} \).

A judgement \( \langle \Pi, \sigma(A) \rangle \) is valid with respect to some variant of SOS if every closed instance of \( \sigma(A) \) is valid with respect to that variant of SOS, and to the program \( \Pi \).

Theorem 5.1.7 (Incompleteness) For some language \( \mathcal{L} \), there is no sound, complete, and finitary proof system \( S \) with program-formula judgements. That
is, for every sound and finitary proof system S with program-formula judgements, there is some such judgement that is valid but not derivable in S.

Proof. Assume that, for every L, there is such a proof system; then prove a contradiction.

Choose L so that we can represent variables, terms, formulae, signed formulae, finite programs, judgements, and derivations for such a proof system, all as closed terms. As Gödel showed, any language with at least one nullary function symbol and at least one non-nullary function symbol suffices. Applying the assumption, we have a sound and complete finitary proof system S for judgements in this language. We can decide on a representation within the language of each expression (variable, term, etc.); let us write [X] for the representation of the expression X.

The readers can convince themselves that we can write a program H containing predicates Subst and Deriv with the following operational properties:

- The closed query Subst(r, s₁, s₂, t) succeeds iff r is some [σ(A)], and we can obtain t from r by substituting all occurrences of [z₁] by s₁ and all occurrences of [z₂] by s₂. (Note that z₁, z₂ are fixed variable names.) Moreover, if r, s₁, s₂ are all closed and t is some variable x, then the query does not fail, and the resultant substitution substitutes a closed term for x.

- The closed query Deriv(r, s, t) succeeds iff s is some [Π], t is some [σ(A)], and r is the representation of a derivation, in the proof system S, of (Π, σ(A)). Moreover, if all the arguments are closed terms, then the query terminates (either succeeds or fails).

(Note that we can write these predicates so that they behave properly under any variant of SOS.)

Now, let U be the signed formula

\[ F(\exists y (\text{Subst}(z₂, z₁, z₂, y) \& \text{Deriv}(x, z₁, y))) \]

Let G be the signed formula U[z₁ := [Π₅], z₂ := [U]]; that is, let G be

\[ F(\exists y (\text{Subst}([U], [Π₅], [U], y) \& \text{Deriv}(x, [Π₅], y))) \]
Is \((\Pi_G, G)\) derivable in \(S\)? If so, then it must be valid, since \(S\) is sound. Therefore \(G\) is valid with respect to \(\Pi\); that is, for any closed term \(t\) we substitute in for \(x\) in \(G\), we have that the query

\[
(\exists y (\text{Subst}([U], [\Pi_G], [U], y) \& \text{Deriv}(t, [\Pi_G], y)))
\]

fails. However, the call to \(\text{Subst}\) cannot fail, and must in fact produce a substitution mapping \(y\) to some closed term \(s\).

But what is this closed term \(s\)? It is in fact \([G]\). From this we must conclude that the query \(\text{Deriv}(t, [\Pi_G], [G])\) fails, for every \(t\). But what this means is that there is no derivation for \((\Pi_G, G)\), even though that was what we just assumed. Thus, \((\Pi_G, G)\) must not be derivable in \(S\) after all.

However, it is still the case that the call to \(\text{Subst}\) cannot fail, and that the call to \(\text{Deriv}\) cannot diverge; so the query (*) must fail, for every choice of \(t\). This means that \((\Pi_G, G)\) is in fact valid but underivable.

This contradicts our first assumption that \(S\) was complete; so for this choice of \(\mathcal{L}\), there can be no sound, complete, and finitary proof system for program-formula judgements.

\(\square\)

The kind of proof systems we have been studying in the other chapters are parameterised by the (implicit) program: that is, there is a different proof system for each program. This parameterisation was done only for convenience, however, and we could instead give calculi whose judgements are sequents with appended programs. It should be clear that the incompleteness proof will still apply to the parameterised proof systems, as long as they are still finitary.
5.2 Infinitary Methods

The incompleteness results of the last section emphasised the restriction to finitary proof systems: proof systems with judgments and derivations which can be represented finitely. Although these are really the only kinds of proof systems which can be used in practice, it is possible to obtain much stronger completeness results by adding infinitary elements to the system.

Of course, if our only concern had been to build a finitary proof system, and we now had decided to remove that restriction, we could simply code the entire operational semantics and definition of validity into a complete "proof system". We must take care to adhere to our original goals of producing as logical a proof system as possible, and not to add elements which are so strong as to make the proof system trivially complete. Infinitary methods meeting these criteria are the addition of an infinitary rule (rule with an infinite number of premisses) to handle free variables in sequents, and the addition of elements to handle divergence by model checking methods.

5.2.1 An Infinitary Rule for Free Variables

In the study of program logics, infinitary rules are often introduced to handle free variables [Gol82]. The technique involves adding a single, simple rule which infers the validity of an assertion from the validity of all its (possibly infinite number of) closed instances. Many useful results follow from this addition - enough to justify the theoretical study of the augmented system.

Consider the following rule:

\[
\text{Inf: } \frac{\Gamma[x := t_1] \rightarrow \Delta[x := t_1]}{\Gamma \rightarrow \Delta} \quad \frac{\Gamma[x := t_2] \rightarrow \Delta[x := t_2]}{\ldots} \quad \Gamma 
\]

where \(t_1, t_2, \ldots\) is an enumeration of all the closed terms in the language

If the language \(\mathcal{L}\) has an infinite number of closed terms, then this rule has an infinite number of premisses. However, the addition of this rule allows us to
remove the restriction to closed sequents in the various completeness proofs. The soundness of the infinitary rule follows immediately from the definition of validity, and we can prove (for instance) the following analogue of the Closed Completeness, stage 4 theorem for PAR:

**Theorem 5.2.1 (Completeness of Infinitary System)** All valid sequents $S$ in which no signed formula appears in a negative context are derivable in LKE+PAR+Inf.

**Proof.** By induction on the total number of free variables which appear in $S$ or $F'$ formulae. Each free variable can be eliminated by one application of Inf. \(\square\)

A more important "characterisation" corollary is the following.

**Theorem 5.2.2** Every instance of a goal formula $A$ succeeds in SOS iff the sequent $[\rightarrow S(A)]$ is derivable in LKE+PAR+Inf; every instance of it fails in SOS iff the sequent $[\rightarrow F'(A)]$ is derivable in LKE+PAR+Inf.

**Proof.** From the Soundness theorem for LKE+PAR+Inf and the Completeness theorem above. \(\square\)

The analogous results also hold for LKE+SEQ+Inf, LKE+PASO+Inf, and any of these systems augmented by the Guaranteed Termination rules of the last section.

The value of this infinitary rule lies in its simplicity, combined with the scope of the resulting completeness. The importance of the infinitary rule is that it says something about the other rules in the given proof systems: it says that they are complete, "except that they cannot handle free variables." This is useful, since it assures us that the rules are not incomplete for some other reason, as for instance they would be if one of the connective axioms were missing. This complements the
knowledge that we have from the Closed Completeness results that the finitary systems are complete for a wide class of closed sequents.

If we had the infinitary rule from the beginning, some of the completeness results would be easier to prove, since we could assume right from the start that all sequents were closed and introduce free variables only at the last stage. Some of the equality rules are redundant in the system with the infinitary rule. In fact, the two groups of equality rules in LKE represent those which would be needed even with unrestricted use of the infinitary rule (Eq, Ineq, and Comp), and those which serve to replace the infinitary rule to some extent in some of the completeness theorems for the finitary systems (Occ, Sub,l, and Sub,r).

Finally, the infinitary rule is very useful in smoothing out a difficulty with the completeness results. Note that the completeness theorem for equality (3.4.3) followed directly from the assumption that there was an infinite number of closed terms in the base language. Without this assumption, the theorem would not be true: there would be a valid sequent satisfying the restrictions in the theorems but underviable in LKE. For example, consider the language with only two terms, the constants \( a \) and \( b \); then the sequent \([\rightarrow x = a, x = b]\) would be valid, since under any of the two minimal substitutions which make it closed ([\(x := a\)] and [\(x := b\)]), one of its formulae is valid. All the other completeness theorems are based on this equality completeness theorem, so all these would fail as well.

However, the case in which there are only a finite number of terms is exactly the case in which the “infinitary” rule is no longer infinitary! Thus in this case we can prove the first stages of the completeness results in a trivial manner by using the “infinitary” rule, and the strong completeness theorem of this section can be proved without infinitary constructs.

**5.2.2 Model Checking for Divergence**

What about the other restriction in the Closed Completeness results, that arose from the inability to handle divergence? Can it be eliminated using infinitary methods?
Chapter 5. Approaches to Incompleteness

It would take us outside the scope of this thesis, but it may be possible to develop a complete proof system based on model-checking techniques [BS90]. This would involve representations of (possibly infinite) models as part of the proof system.

There is also a body of literature on detecting infinite loops in Prolog programs, with varying degrees of formality of approach [PG85,vG87,ABK89]. It may be possible to formalise one of these approaches within the context of the sequents defined here in order to obtain a finitary and more complete proof system.

5.3 Induction

The finitary proof systems of the last chapters are complete for all closed sequents with empty antecedents, and all sequents involving non-recursive predicates. However, there are many practically important non-closed assertions involving recursive predicates which cannot be proven by them; and needless to say, the infinitary techniques of the last section are not useful in practice.

The main finitary method of handling such assertions in practice is induction. This is a generalisation of the usual practice of proving a statement about the natural numbers using "mathematical induction": if the statement holds for 0, and whenever it holds for \( n \) it holds for \( n + 1 \), then it holds for all natural numbers.

In the context of proof systems, we can add proof rules which formalise this method of reasoning. The two major types of induction rules that we might add are the simpler, but less general subterm induction rule, and the more complex but much more general and useful well-founded induction rule. Neither of these rules give us a complete proof system, because of Gödel incompleteness; but they do enable us to prove a useful subset of the non-closed sequents.
5.3.1 Subterm Induction

The proof rules formalising what I call “subterm induction” allow us to do only induction over the structure of terms. This is a common form of inductive reasoning about programs, first noted by Burstall [Bur69]; it can be seen as a formal justification of such programming techniques as CDR’ing down a list, or tree traversal.

To introduce this idea, let us see how induction can be done in the context of the natural numbers. Takeuti [Tak87] gives the following induction rule for natural numbers (in our notation):  

\[
\Gamma, A(x) \rightarrow A(s(x)), \Delta \\
\Gamma, A(0) \rightarrow A(t), \Delta
\]

where \( x \) does not appear free in \( \Gamma, \Delta \) or \( A(0) \).

This rule can be read as saying that if (a) whenever a formula \( A \) is true of some term \( t \) it is true of \( s(t) \), and (b) \( A \) is true of \( 0 \), then we can conclude that it is true of every term. We can prove this rule sound, in the context of arithmetic, because the only basic term constructors we have are the constant \( 0 \) and the unary function symbol \( s \). If we had more function symbols, the rule would not be sound because there is no guarantee that \( A \) is true for (for instance) some other constant.

One generalisation of this rule to take in arbitrary languages can be formulated as follows.

**Definition 5.3.1** Let \( S \) be of the form \( \Gamma \rightarrow \Delta, \sigma(A(x)) \), where \( A(x) \) is the only formula in \( S \) to have \( x \) free. Let \( f \) be an \( n \)-ary function symbol. Then \( IH(S, f, x) \) is the sequent  

\[
\Gamma, \sigma(A(y_1)), \ldots, \sigma(A(y_n)) \rightarrow \Delta, \sigma(A(f(y_1, \ldots, y_n)))
\]

where \( y_1, \ldots, y_n \) are \( n \) variables not free in \( S \).

The induction rule (or more properly rule schema) for the sequent calculus now takes the form  

\[
\frac{IH(S, f_1, x) \ldots IH(S, f_m, x)}{S}
\]
where $S$ is any sequent of the form $\Gamma \rightarrow \Delta, \sigma(\Delta(x)), \Delta(x)$ is the only formula in $S$ to have $x$ free, and $f_1 \ldots f_m$ are all the function symbols in the language $L$. This rule, which I will call SUBTERM, is sound in the same sense that the other rules we have discussed are sound, as stated by the following theorem.

**Theorem 5.3.2** If each premiss of an application of the induction rule is valid with respect to some variant of SOS, then the conclusion is valid with respect to it.

**Proof.** If $\Gamma \rightarrow \Delta$ is valid, then the result holds trivially. Otherwise, there is some substitution $\theta[x := t]$ under which all of the $\Gamma$ signed formulae are valid, and we must prove that $\sigma(\Delta)$ is valid under it. We can do this by induction on the structure of $t$; each subcase corresponds to one of the premisses of the rule application.

$\square$

See Section A.4 for an extended example of the use of subterm induction – a derivation of the sequent $[-4 S(Add(x, 0, x))]$. It is possible to derive this sequent using the infinitary rule mentioned in the last section, but not with just the rules in the finitary proof systems of Chapters 3 and 4. Subterm induction allows us to derive it using only finitary methods.

### 5.3.2 Disadvantages of Subterm Induction

The advantage of subterm induction is that the rule is easily stated, and easily applied for simple languages. As the language increases in size, however, we begin to run into problems.

If we had other function symbols in the language – say the constant symbol $a$ – example sequents such as the one above would not even be valid, because $Add(a, s, t)$ fails for every $s$ and $t$. What would be valid is the sequent $[\rightarrow S(N(x)) \supset S(Add(x, 0, x))]$, where $N$ is the predicate which checks whether its parameter is a Peano natural number:

$$N(x) \leftrightarrow x = 0 \lor \exists px(x = s(px) \& N(px))$$
This latter sequent we could prove using the induction rule. However, the premiss for the constant $a$ would be $\forall S(N(a)) \supset S(Add(a,0,a))$, which would be valid only by virtue of $N(a)$ failing.

In general, as we add function symbols, we add premisses to the induction rule which we must prove in many trivial cases. This is a problem in general with logic programs, because we often want to prove properties about predicates while assuming some type information about the parameters to the predicate (e.g., that all parameters are lists), but we generally have many function symbols.

Another shortcoming of subterm induction is that not all recursions found in programs are on the subterm structure of the parameters to a predicate. Some predicates terminate in general because of some other decreasing measure. For assertions involving these predicates, we could define a natural number measure on the parameters and use subterm induction on natural numbers to derive the relevant sequents. This may turn out to be rather complicated, however; a more general solution is discussed in the next section.

If we were working with a strongly typed logic, in which all formulae, terms and variables have an associated type, some of these problems would vanish. In typed logic programming languages (and strongly typed programming languages in general [CW85]), the definition of a predicate includes a specification of the types of its arguments; calls to predicates with arguments of anything other than the correct types are syntactically ill-formed. The number of cases to consider in the induction rule would therefore not increase with bigger languages; for instance, one would specify that the arguments to $Add$ were exclusively of natural number type, and only two premisses would ever be needed for the induction rule.

The addition of types to our language would take us outside the scope of this thesis, although some form of typing is clearly desirable, for this and other reasons. However, even in a typed language we are left with the problem of recursions which do not act on subterm structure. This problem must still be handled by a generalisation of induction to induction on any well-founded measure.
5.3.3 Well-Founded Induction

General well-founded induction [MW78] is more powerful (in the sense of allowing more proofs) and in some ways simpler than subterm induction. With well-founded induction, there is only one premiss to the induction rule, although this premiss depends on a metatheoretic proof of a property of a predicate.

Manna and Waldinger's formulation, from which this discussion is drawn, uses relations as part of the program logic. Because we are talking about logic programming, we can use object logic relations (the defined predicates of the program) as our well-founded order relations.

Definition 5.3.3 A predicate \( R \) defined in a program \( \Pi \) is a well-ordering in \( \Pi \) if:

- For no closed \( t \) does \( R(t, t) \) succeed; and

- There is no infinite sequence of closed terms \( t_1, t_2, t_3, \ldots \) such that \( R(t_{i+1}, t_i) \) succeeds for all \( i \geq 1 \) ("there is no infinite descending sequence of closed terms").

Using well-ordering predicates, we can define an induction rule, WF, which is in some ways simpler and more useful than that of the last section:

\[
\Gamma, \forall x (S(R(x, y)) \supset A(x)) \rightarrow A(y), \Delta \quad \Gamma \rightarrow A(y), \Delta
\]

where \( R \) is a well-ordering and \( y \) does not appear free in \( \Gamma \) or \( \Delta \). (Note that the proof that \( R \) is a well-ordering cannot be done within the proof system, and must be done metatheoretically.)

The definition of well-ordering could be made more straightforward by our insisting that, for instance, \( R(t, t) \) fail. However, the weaker conditions above suffice to ensure the soundness of the well-founded induction rule.

Theorem 5.3.4 If the premiss of an application of the well-founded induction rule is valid (with respect to some variant of SOS), then the conclusion is valid (with respect to that variant of SOS).
Proof. By reductio ad absurdum. Call $t$ a counterexample for an assertion $B(x)$ with free variable $x$ if $B(t)$ is invalid.

Assume the premiss is valid but the conclusion is invalid. (This is the reductio assumption.) This means that there is a $\theta[y := t_1]$ which makes the conclusion closed, such that all its antecedent assertions are valid under $\theta[y := t_1]$ but none of its consequent assertions (including $A(y)$) is valid under $\theta[y := t_1]$. (In other words, $t_1$ is a counterexample for $A(y)\theta$.) However, by assumption, the premiss is valid; so since none of its consequent assertions are valid under $\theta[y := t_1]$, one of its antecedent assertions must be invalid under $\theta[y := t_1]$; and the only one that can be is $\forall x (S(R(x,y)) \supset A(x))$. Since this is invalid under $\theta[y := t_1]$, there must be a $t_2$ such that $(S(R(x,y)) \supset A(x))[x := t_2]\theta[y := t_1]$ is invalid; that is, such that $R(t_2, t_1)$ succeeds but $(A(t_2))\theta$ is invalid.

But this would mean that $t_2$, like $t_1$, is also a counterexample for $A(y)\theta$, and since $R$ is a well-ordering, $t_2$ must be distinct from $t_1$. So we can follow the same line of reasoning to get another counterexample $t_3$ such that $R(t_3, t_2)$ succeeds, and so on. But then we will have the infinite descending sequence of closed terms which we denied we had when we assumed $R$ was a well-ordering. Contradiction; so the conclusion of the application of the rule must be valid after all.

$\square$

With well-founded induction, we can prove all of the things we could with subterm induction (since the subterm inclusion predicate is a well-ordering), but in some situations the derivations are simpler. Consider the sequent $[\neg S(N(x)) \supset S(Add(x,0,x))]$, which was easy to prove with only two function symbols in the language, but became more cumbersome as the number of symbols increased. With well-founded induction, the complexity of the derivation is large, but the derivation is independent of the number of function symbols in the language. One possibility for a well-ordering predicate is the predicate $R$ given by the following definitions:

$$R(x,y) \leftrightarrow N(x) \& N(y) \& Lt(x,y)$$

$$Lt(x,y) \leftrightarrow \exists py(y = s(py) \& (x = py \lor Lt(x,py)))$$
In the new derivation, the assumption in \( R \) that \( x \) is a natural number cancels the assumption \( N(x) \) in the assertion to be proved, and the derivation has no more branches than did the subterm induction derivation. This derivation is also in the Appendix.

The main disadvantage of well-founded induction is that the order relation must be formulated, proven well-founded, and (in our formulation) defined in the program. This is quite tedious, especially if the order relation is of the simple kind given in the example above. There should be some way of automatically generating order relations from given type assumptions which will allow us to prove assertions involving those type assumptions. As with the corresponding problem with subterm induction, an explicit type system in our logic would help here.
Chapter 6

Summary and Future Directions

This thesis has taken as its object of study the control-discipline variants of a simple logic programming language equivalent to Horn clause logic programming. It has classified and logically characterised the set of successful and failing queries of these variants of the language.

I have given an operational semantics, SOS, of which variants correspond to the parallel "and" and "or", sequential "and", sequential "or", and sequential "and" and "or" control disciplines. This operational semantics homogenises the treatment of the control disciplines by incorporating control information (such as the failure-backtrack mechanism of sequential systems) into the operational semantics. (Some of the variants of SOS have equivalent compositional operational semantics, which I have given.) I have also classified the queries into those succeeding and those failing in each of the control disciplines, and have proven the equivalence of some of these classes.

I have then used a sequent calculus framework, in which the elements of sequents are assertions about the success or failure of queries, to give a logical analysis of these classes of queries. Three calculi are given; they share a common set LKE of rules for classical logic with equality as syntactic identity, and differ in the set of axioms which characterise the behaviour of queries.

- LKE+PAR characterises the queries which succeed in parallel-or systems, and those which fail in parallel-and systems;
- LKE+SEQ characterises the queries which succeed in the sequential-and, sequential-or system, and those which fail in sequential-and systems;

- LKE+PASO characterises the queries which succeed in the parallel-and, sequential-or system.

The precise sense in which these calculi "characterise" the classes of queries is that if a query succeeds or fails in a particular control discipline, the corresponding assertion of its success or failure is derivable in the appropriate calculus. The value of these characterisations is that they give a precise, logical account of which queries fail or succeed.

These calculi can also be used for proving more general properties of logic programs, including general termination and correctness properties. This is important, as it addresses issues of program correctness and specification which arise in practical settings. The sequent calculi can therefore act as a basis for practical tools for proving such properties of programs. See below for a more detailed discussion of this possibility.

The sequents of the calculi are sufficiently expressive that the calculi cannot be complete with respect to the natural notion of validity; but I have shown several results that give a wide range of valid, derivable sequents. I have also analysed the senses in which the systems in question must be incomplete, and have given extensions, such as induction rules, which can be used to prove a wide class of useful properties of programs.

### 6.1 Language Extensions

One of the main deficiencies of this thesis, from a practical point of view, is that the programming language it considers is not very expressive or powerful. Logic programs as I have defined them are a variant of the strict Horn clause programs originally described by Kowalski [Kow74]. Since then, the state of the art has moved on considerably. It would be desirable to incorporate more recent
developments into the language being characterised, to see whether they can be treated in the proof-theoretic framework given in this thesis.

6.11 Iterated Implication and Universal Quantification

As Gabbay and Reyle [GR84] and Miller and Nadathur [MN86] have shown, it is possible to add implication and universal quantification to the class of goal formulae, in restricted contexts, and get a logic programming language which has a reasonable operational semantics. The resulting language allows many useful programming constructs, including the incremental buildup of the definition of a predicate, module definitions, and variable scoping. For instance, if $D_1, D_2$ and $D_3$ are conjunctions of predicate declarations and $G_1$ and $G_2$ are goals, then the goal $D_1 \supset (D_2 \supset G_1) \& (D_3 \supset G_2)$ indicates that the system is to search for a solution to $G_1$ in the context of the program $D_1 \& D_2$, and for a solution to $G_2$ in the context of the program $D_1 \& D_3$.

Can languages such as that specified by hereditary Harrop formulae (or languages with similar iterated implication and universal quantification constructs) be analysed within the proof-theoretic framework? It may be possible, but it may require a more complex form of signed formula, and the treatment of sequentiality may be more problematic.

To give a proof-theoretic characterisation similar to those in this thesis, it seems that signed formulae must be of the form $\sigma^\Pi(A)$, where $\Pi$ is a program, since the program can change during the course of a computation. Because of the possible incremental buildup of the program, we must abandon the bi-implicative form of predicate definition, and stay with the definition of a program as a hereditary Harrop definition.

The operational semantics given by Miller et al. [MN86] is nondeterministic, and so specifies a parallel discipline; for a sequential interpreter, one would have to assume something about the order in which clauses are added to the program (at the end?), and possibly construct the Clark completion of a predicate definition whenever a predicate expansion is performed.
6.1.2 Types

Types are important in any programming language. Their main practical use is to allow programmers to specify what kinds of arguments they expect predicates (or functions or procedures) to take, so that the compiler or interpreter can inform programmers of inconsistencies in use of predicates. In large programs, this can be very helpful, as this kind of type inconsistency is a common source of errors and can be checked relatively cheaply.

Types would also be important in facilitating inductive proofs of properties of logic programs. As suggested in Section 5.3, it may be possible to give simpler rules for subterm induction over given types, and the presence of types can considerably simplify the statements of theorems to be proven (as all type assumptions are implicit in the predicate applications being made). To do induction over the structure of a given free variable, one would choose the appropriate rule for that variable, eliminating the need for vacuous premisses corresponding to incorrectly-typed terms and reducing the need for well-ordering predicates for well-founded induction.

The question of how we should add types to logic programming languages is an area of current research [PR89,Red88]. An important question would be which of the currently-proposed type systems would be best suited to proof-theoretic analysis.

6.1.3 Variable Modes

In logic programming, variable mode declarations on predicate arguments are arguably as important as type declarations. A variable mode declaration [DW83] is a declaration of how instantiated the programmer expects an argument to be when the predicate is called or when it returns; examples of modes include “var” (the parameter is an unbound variable), “nonvar” (the parameter may be partially instantiated), and “ground” (the parameter is fully instantiated).
Like type declarations, mode declarations can be used by programmers to enlist the help of the compiler or interpreter to check that a predicate is used as expected. However, there are other uses of modes. Modes allow us to optimise unifications in a goal or program; for instance, in a computation of $x = y$ where $y$ is of "ground" mode, the unification is at worst a matching (for which very fast algorithms are known) and at best, when $x$ is of "var" mode, a simple assignment. Modes thus also allow us to identify classes of predicates which can be compiled or interpreted efficiently. These ideas are implemented in Paul Voda's language Trilogy [And87].

As might be suggested by the completeness proofs in this thesis, information that a variable represents a ground term is useful in proving assertions. Plümer [Plü90b, Plü90a] has used mode information to prove termination of logic programs, for the more limited notion of termination in which all solutions can be found finitely. Some of his techniques might be formalisable within the framework of a sequent calculus.

Finally, when programmers have the ability to specify modes, language designers have the ability to insist that only variables with certain modes be used in certain contexts. As we will see below, this can be used to add new features to the language, such as negation, which can be implemented in a more efficient and complete manner with mode information.

Other modes, and variations on the definition of mode, have been given in the literature [DW83, Dev90]. Which definitions would go best with the proof-theoretic technique is a topic which deserves investigation.

6.1.4 Predicate Names as Terms

Several extensions to logic programming have been proposed which move it into the realm of higher order logic [MN86, CKW89]. I have explored one such simple extension [And89a], in which predicate names are treated as atomic terms, in a simpler setting than the one in this thesis; viewing it in a sequent calculus setting raises interesting questions.
Chapter 6. Summary and Future Directions

The extension is based on Gilmore's nominalistic set theory NaDSet [Gil86]. Besides treating predicate names as terms, the extension allows any term to appear where only a predicate name can appear in first order logic programming. (It also makes several other extensions which are unimportant for this discussion.) The operational semantics given in [And89a] is nondeterministic, that is, parallel-and and parallel-or; the proof-theoretic characterisation is a natural deduction proof system for successful queries. These systems should be easy to cast into the parallel SOS and sequent calculus framework.

However, when we consider issues of sequentiality, problems arise. Consider the query \( \exists x(x(3) \& false \& x = P) \), where \( P \) is some predicate name. The nondeterministic (parallel) operational semantics fails on this query. If we sequentialised the operational semantics in the natural way – computing the conjuncts from left to right – the query would fail in the sequential system too. Neither system would evaluate the query in such a way as to require that \( P(3) \) be computed, instead deferring the computation until \( x \) is bound or the query fails. However, we cannot use the rules for failing existential and conjunctive queries from SEQ here, because that would require a proof that \( (P(3) \& false \& P = P) \) fails – which may not be true in the sequential system since \( P \) might diverge.

This problem has some of the flavour of the disjunction problem which caused us to introduce disjunctive unfoldings. It may therefore be solvable by a similar technique. It may, however, be easier to add this feature to a language which already has variable mode declarations. Then, we could restrict occurrences of variables in predicate positions to those declared to be of “ground” mode. This would exclude the query given above, but still include many of the useful predicates which we want to write in a higher order setting.

6.1.5 Negation and If-Then-Else

Negation was one of the first features that researchers attempted to add to Horn clause logic programming, and has remained one of the most problematic. In the proof-theoretic setting, we would naturally like to have axioms like \( S(\neg A) \leftrightarrow F(A) \)
and \( F(\neg A) \leftrightarrow S(A) \). Whether we can achieve this depends strongly on the operational treatment of negation.

I will not attempt to review the extensive work done on the semantics of negation in Prolog. I will simply point out that not all of the work on negation in sequential languages would necessarily mesh well with the analysis in this thesis. For instance, in Naish's MU-Prolog [Nai84], the computation of non-ground negated subgoals is delayed until they are ground, or until there are only non-ground negated subgoals left. Essentially, subgoals are re-ordered as the computation progresses. This strategy, while practically very useful, would play havoc with attempts at giving a logical characterisation of the exact set of failing and successful queries along the lines of this thesis.

As with predicates-as-terms, variable mode declarations might enable us to give a reasonable account of negation. We could insist that all variables appearing in negated goal formulae be of "ground" mode, and use negation as failure for the allowed occurrences of negation. In this way, we can capture a large class of uses of negation without sacrificing logicality; as Jaffar et al. have shown [JLL83], negation as failure is a complete strategy for ground negated queries.

Some form of if-then-else construct would also be possible if we used modes in this way. The expression if \( A \) then \( B \) else \( C \) could be defined as being equivalent to \( (A \& B) \lor(\neg A \& C) \), if we maintain the syntactic restriction of \( A \) to having only input-mode free variables.

In an if-then-else formula, the condition \( A \) need be computed only once. It is therefore better to have an explicit if-then-else construct than to just write the equivalent formula, since it not only clarifies the underlying meaning of the formula, but also signals to the compiler or interpreter that this optimisation can be done. A bonus of such an if-then-else construct is that it can be used to replace many uses of the non-logical Prolog "cut", which is commonly used to avoid having to compute the negation of a positive condition in another clause.
6.1.6 Constraints

The area of constraints seems promising as a possible extension to the framework in this thesis. This is because constraint logic programming languages generally extend basic logic programming in ways which are independent of the various control disciplines.

One description of a subset of the constraint logic programming languages is as follows. We are given a first order language $\mathcal{L}$, with some predicate names in $\mathcal{L}$ being identified as constraint predicates. We are also given a semantics for these constraint predicates – a characterisation of which applications of the constraint predicates are considered to be true and which false. (How many of the useful CLP languages described by Jaffar and Lassez [JL86] this description captures is not clear.)

In conventional logic programming (which is subsumed by the constraint paradigm), the only constraint predicate is equality, which is interpreted as syntactic identity. But other systems are possible: for instance, the addition of $\neq$ as a constraint predicate [And86], the interpretation of $=$ as identity between infinite rational trees [CKvC83], Presburger arithmetic, and linear arithmetic [JL86].

In the sequent-calculus framework of this thesis, none of these languages would seem to require anything more than an axiomatisation of the semantics of the constraint predicates, along the lines of the axiomatisation of equality in LKE. On the operational semantics side, we need, for each constraint language, a unification (or "subsumption") algorithm which allows us to completely decide whether a given finite set of constraint predicate applications is satisfiable or not.

Because of the requirement that a subsumption algorithm exist, one would expect that the entire analysis of successful and failing queries for both parallel and sequential systems would go through as in this thesis. In particular, there would be analogues of the Equality Completeness theorem (3.4.3) which would form the basis of the rest of the completeness theorems; the remainder of the theorems are mainly about the connectives and defined predicates, and so would go through as before.
6.1.7 Multiple Control Disciplines

As mentioned in Chapter 2, most practical languages which have parallel "and" and "or" also have the sequential versions of these connectives. It would be useful to have a language in which all four connectives (parallel and sequential "and" and "or") were represented, and the associated operational semantics and proof theory.

The proof theory would seem to be straightforward; it should be just the amalgamation of all of the rules from PAR and SEQ, with the success axioms for conjunction from PAR becoming axioms for the parallel "and", and so on. What would be more difficult is the operational semantics. The interaction of the four connectives is sufficiently complex that it does not seem to fit easily into any of the styles of semantics given in Chapter 2. Voda [Vod85] has successfully given an operational semantics for such a language in a multi-step style with "tags" indicating current points of computation. However, this operational semantics is sufficiently different from those given in this thesis that it would require a very different structure of soundness and completeness theorems.

6.2 Practical Program Proving

Program proving, the practice of proving useful properties of programs, has been studied for many years. I have made some reference to practical program proving in the course of this thesis, particularly in the section on induction; although optimisations would be needed for converting the proof systems given here into a practical tool for proving properties of logic programs, those proof systems do act as a basis for such a tool.

Most past program-proving research has gone on in the paradigms of imperative or functional programming. I feel that this may have been, in part, a result of the split between theory and practice in logic programming. Practical program proving is intended to apply to practical languages; the practical languages in
logic programming are the sequential ones; and, until now, there has been no satisfactory logical account of sequential logic programming. The closest we have come to program proving of sequential logic programs has been the operational work of Francez et al. [FGKP85] and Baudinet's translation of logic programs into functional programs to be proven with known functional tools [Bau88]. The contributions in this thesis go some way towards filling this gap between theory and practice.

Past experience in other paradigms can help research into logic program proving. But besides the usual scientific motivations of wanting to apply given knowledge to a new area of study, there are important considerations to do with methodology of programming in this area of research.

Logic programs are often compared to "executable specifications"; the logical outlook we need to write logic programs carries over to the process of proving program properties. It may be, therefore, that it would be easier to prove properties of logic programs than of imperative or functional programs, if we had the appropriate tools. Tools to assist or automatically generate proofs of logic programs would serve to test this hypothesis and investigate its consequences.

Tools for proving properties of logic programs would not necessarily be very useful by themselves. They would be more useful in the context of a comprehensive logic program development environment. Mello and Natali [MN89] discuss such an environment, Contextual Prolog, which lacks a theorem prover but provides the necessary tools for refinement of prototype programs. It would be interesting to see how this or other approaches to development could benefit from a proof assistant system.
Bibliography


145

[Bibliography]


Appendix A

Examples

A.1 Conventions

Computations in this appendix of examples will appear in the following format: with the starting backtrack stack on a line by itself, and then each step on a separate line or lines consisting of the number of the computation rule from the relevant SOS variant, the production arrow, and the resultant backtrack stack. For example, in the computation

\( (((): (\text{Loop}() \& \text{false}) \lor \text{true}) \)

(2) \( \overrightarrow{\text{SOS}/\text{so}} (((()) : \text{Loop}() \& \text{false}); ((()) : \text{true}) \)

(1) \( \overrightarrow{\text{SOS}/\text{so}} (((()) : \text{Loop}(), \text{false}); ((()) : \text{true}) \)

... the second backtrack stack is derived from the first by an application of rule 2 (\( \lor \)) from \( \text{SOS}/\text{so} \); the third is derived from the second by an application of rule 1 (\( \& \)) from \( \text{SOS}/\text{so} \); and so on.

Due to formatting difficulties, the example derivations in this appendix will not be of the tree format in which the rules are given. They will be instead in the following style: each sequent will be given on a separate line, with the premisses in the derivation of each sequent being above it and indented. Dots will be placed on each line to help the reader see the indentation.

Thus the derivation which is written in tree style as
Appendix A. Examples

These examples are based on the definition of the \(\text{Mem}\) predicate:

\[
\text{Mem}(x,l) \leftrightarrow \exists h \exists t (l = [h|t] \& (x = h \lor \text{Mem}(x,t)))
\]

The query we will be concerned with is \(\text{Mem}(x,[a|[b|[ ]]])\). I assume that \([\ ]\) is the nullary "empty list" constant and \([ [ ]\) is the binary list formation function symbol, and that \(a\) and \(b\) are constants. I will give the SP (sequential) computation of this query, then an alternate SOS computation which makes use of the parallel "or"; then I will give two derivations in LKE+PAR and two in LKE+SEQ for the query.

A.2.1 Computations

The SP computation for the query is in Figure A-1. The computation ends with the solution \(a\) for \(x\).

One possible SOS-computation starts off the same, but then proceeds differently after the second-last step by expanding the predicate call in the second
Appendix A. Examples

closure, ending with the solution $b$ for $x$. This computation is given in Figure A–2.

Note the renaming that goes on in the second and third steps, the first renaming (of $t$ to $t_1$) to avoid variable capture arising from the definition of substitution, and the second (from $h$ to $h_1$) arising from the $\exists$ rule.

A.2.2 Derivations

The sequent corresponding to the statement that the query succeeds is the following one:

$$\rightarrow S(\exists x Mem(x, [a[[b]]]))$$

In PAR, one possible derivation is given in Figure A–3. Only the essential steps are given; in particular, the major premisses of the numerous Cut rules are omitted. They can be easily guessed by inspection of the PAR axioms.

Another derivation, which corresponds to the second computation given above, is given in Figure A–4. The difference arises here only from a different choice of witness for $x$.

The more complex SEQ derivation is given in Figure A–5. Note that we must do a predicate and then a disjunctive unfolding to handle the predicate call in the query.

A.3 Infinite Loop

Here, we treat the slightly paradoxical example of the query which succeeds in all operational semantics except SP, but diverges in SP. The example is the query

$$(Loop() \& false) \lor true$$

where $false \equiv 0 = 1$, $true \equiv 0 = 0$, and $Loop$ is defined with the predicate definition

$$Loop() \leftrightarrow Loop()$$
Appendix A. Examples

Figure A-1: The computation of the query $\text{Mem}(x, [a[b[[ ]]])$ in SP.

... 

Figure A-2: One possible computation of the query $\text{Mem}(x, [a[b[[ ]]])$ in SOS; the first step shown is the second last of the SP computation.
Appendix A. Examples

\[ S([a][b[ ]] = [a][b[ ]]) \]
\[ S(a = a) \]
\[ S(a = a \lor \text{Mem}(a,[b[ ]])) \]
\[ S((a|[b[ ]] = [a][b[ ]] \land (a = a \lor \text{Mem}(a,[b[ ])))) \]
\[ S(\exists t \,(a|[b[ ] ] = [a][t] \land (a = a \lor \text{Mem}(a,t)))) \]
\[ S(\exists h \exists t \,(a|[b[ ] ] = [h][t] \land (a = h \lor \text{Mem}(a,t)))) \]
\[ S(\text{Mem}(a,[a][b[ ]])) \]
\[ S(\exists x \text{ Mem}(x,[a][b[ ]])) \]

Figure A-3: A derivation of the sequent \[ \rightarrow S(\exists x \text{ Mem}(x,[a][b[ ]])) \] in LKE+PAR. Only essential steps are given.

\[ S([a][b[ ]] = [a][b[ ]]) \]
\[ S([b[ ]] = [b[ ]]) \]
\[ S(b = b) \]
\[ S(b = b \lor \text{Mem}(b,[ ])) \]
\[ S([b[ ]] = [b[ ]] \land (b = b \lor \text{Mem}(b,[ ]))) \]
\[ S(\exists t \,(b|[ ] = [b][t] \land (b = b \lor \text{Mem}(b,t)))) \]
\[ S(\exists h \exists t \,(b|[ ] = [h][t] \land (b = h \lor \text{Mem}(b,t)))) \]
\[ S(\text{Mem}(b,[b[ ]])) \]
\[ S(\exists x \text{ Mem}(x,[a][b[ ]])) \]

Figure A-4: Another derivation of the sequent \[ \rightarrow S(\exists x \text{ Mem}(x,[a][b[ ]])) \] in LKE+PAR. Only essential steps are given.
The computation of this query in SOS/so is in Figure A–6.

There is no terminating SP computation of the query; the computation sequence just loops on the predicate call. See Figure A–7 for this computation.

A.3.1 Derivations

To see that \( (\text{Loop}() \& \text{false}) \lor \text{true} \) succeeds in SOS (or SOS/sa, since its set of successful queries is the same), we need only give the derivation for the sequent \( \rightarrow S((\text{Loop}() \& \text{false}) \lor \text{true}) \) in \( \text{LKE+PAR} \). This derivation is in Figure A–8.

It is only slightly more complex to prove that it succeeds in SOS/so, by giving the \( \text{LKE+PASO} \) derivation for the query. (Recall that PASO takes the success rules from SEQ and the failure rules from PAR.) See Figure A–9 for this derivation.

However, as we would expect, it is impossible to give a derivation of the sequent in SEQ. Any such derivation would involve a derivation of \( \rightarrow F(\text{Loop}() \& \text{false}) \), which can be proven in turn only by proving \( \rightarrow F(\text{Loop}()) \). This is clearly impossible.

A.4 Subterm Induction

The example to be proven here is the sequent

\[ \rightarrow S(\text{Add}(x, 0, x)) \]

that is, “for all terms \( t \), the query \( \text{Add}(t, 0, t) \) succeeds”, where \( \text{Add} \) is defined as follows:

\[
\begin{align*}
\text{Add}(x, y, z) & \iff (x = 0 \& y = z) \\
& \lor \exists px, pz (x = s(px) \& \text{Add}(px, y, pz) \& z = s(pz))
\end{align*}
\]

This is true only if we make some assumptions about the language. We will assume that the language contains only the unary function symbol \( s \), and the
Appendix A. Examples

. . . . . . . . . . → S([a][b[ ]] = [a][b[ ]])
. . . . . . . . . . → S(a = a)
. . . . . . . . . . → S(([a][b[ ]] = [a][b[ ]]) & a = a)
. . . . . . . . . . → S(∃t ([a][b[ ]] = [a][t]) & a = a)
. . . . . . . . . . → S(∃h ∃t ([a][b[ ]] = [h][t] & a = h))
. . . . . . . . . . → S(∃x ∃h ∃t (B))
. . . . . . . . . . → S(∃x ∃h ∃t (B) V (F(∃x ∃h ∃t (B)) & S(∃x ∃h ∃t (C))))
. . . . . . . . . . → S(∃x ∃h ∃t (B) V (∃x ∃h ∃t (C)))
. . . . . . . . . . → S(∃x ∃h (∃([a][b[ ]] = [h][t] & (x = h V Mem(x, t))))
→ S(∃x Mem(x, [a][b[ ]])))

where B ≡ ([a][b[ ]] = [h][t] & x = h) and C ≡ ([a][b[ ]] = [h][t] & Mem(x, t))

Figure A-5: A derivation of the sequent [→ S(∃x Mem(x, [a][b[ ]]))] in LKE+SEQ. Only essential steps are given.

((0) : (Loop() & false) V true)

(2) SOS/so ((0) : Loop() & false); ((0) : true)
(1) SOS/so ((0) : Loop(), false); ((0) : true)
(6) SOS/so ((0) : true)
(5) SOS/so ((0) :)

Figure A-6: Computation of query (Loop() & false) V true in SOS/so.

((0) : (Loop() & false) V true)

(2) SP ((0) : Loop() & false); ((0) : true)
(1) SP ((0) : Loop(), false); ((0) : true)
(4) SP ((0) : Loop(), false); ((0) : true)
(4) SP ((0) : Loop(), false); ((0) : true)
(4) SP ...

Figure A-7: First part of infinite computation sequence of query (Loop() & false) V true in SP.
Appendix A. Examples

\[ 0 = 0 \]
\[ S(\text{true}) \]
\[ S((\text{Loop()}) \& \text{false}) \lor \text{true} \]

Figure A-8: Derivation of \([\rightarrow S((\text{Loop()}) \& \text{false}) \lor \text{true}]) in LKE+PAR. Only essential steps are given.

\[ 0 = 1 \rightarrow \]
\[ \rightarrow \neg 0 = 1 \]
\[ \rightarrow F(\text{false}) \]
\[ \rightarrow F(\text{Loop()}) \lor F(\text{false}) \]
\[ \rightarrow F(\text{Loop()}) \& \text{false} \]
\[ \rightarrow 0 = 0 \]
\[ \rightarrow S(\text{true}) \]
\[ \rightarrow F(\text{Loop()}) \& \text{false} \& S(\text{true}) \]
\[ \rightarrow S(\text{Loop()}) \lor (F(\text{Loop()}) \& \text{false}) \& S(\text{true})) \]
\[ \rightarrow S((\text{Loop()}) \& \text{false}) \lor \text{true} \]

Figure A-9: Derivation of \([\rightarrow S((\text{Loop()}) \& \text{false}) \lor \text{true}]) in LKE+PASO. Only essential steps are given.
constant (nullary function symbol) 0. See Figure A–10 for the derivation of this sequent in LKE+PAR+SUBTERM.

A.5 Well-Founded Induction

The example to be proven here is the analogue of the sequent in the last section:

$$\rightarrow S(N(y)) \supset S(Add(y, 0, y))$$

that is, “for all terms t such that N(t) succeeds, the query Add(t, 0, t) succeeds”.

The definitions of N, the predicate testing for whether its argument is a Peano natural number, the order predicate, R, and its auxiliary predicate Lt, are as follows:

$$N(x) \leftrightarrow x = 0 \lor \exists px(x = s(px) \& N(px))$$

$$R(x, y) \leftrightarrow N(x) \& N(y) \& Lt(x, y)$$

$$Lt(x, y) \leftrightarrow \exists py(y = s(py) \& (x = py \lor Lt(x, py)))$$

We now need no assumptions on the language (except, obviously, that it contains 0 and s). See Figures A–11 and A–12 for the derivation of this sequent in LKE+PAR+WF.
Figure A-10: Derivation of $\rightarrow S(Add(x,0,x))$ in LKE+PAR+SUBTERM. Only essential rule applications are shown.
Appendix A. Examples

159

• (as in subterm induction example)

                               \[ \rightarrow S(Add(0,0,0)) \]

\[ B(0), \rightarrow S(Add(0,0,0)) \]

\[ B(y), S(y = 0) \rightarrow S(Add(y,0,y)) \]

          ... (see next figure)

\[ B(y), S(y = \text{s}(py)&\text{N}(py)) \rightarrow S(Add(y,0,y)) \]

\[ B(y), S(\exists px(y = \text{s}(px)&\text{N}(px))) \rightarrow S(Add(y,0,y)) \]

\[ B(y), S(y = 0 \lor S(\exists px(...)) \rightarrow S(Add(y,0,y)) \]

\[ B(y), S(N(y)) \rightarrow S(Add(y,0,y)) \]

\[ B(y) \rightarrow C(y) \]

\[ \rightarrow C(y) \]

where \( C(z) \equiv S(N(z)) \supset S(Add(z,0,z)) \) and \( B(z) \equiv \forall x(S(R(x,z)) \supset C(z)) \)

Figure A-11: Derivation of \( \rightarrow S(N(y)) \supset S(Add(y,0,y)) \) in LKE+PAR+WF. The derivation of \( B(y), S(y = \text{s}(py)&\text{N}(py)) \rightarrow S(Add(y,0,y)) \) is in the next figure.
Appendix A. Examples

.. . . . . . \( S(N(py)) \rightarrow S(N(py)) \)

. . . . . . . . . . . . . (straightforward)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(N(s(py))) \)

. . . . . . . . . . . . . (straightforward)

. . . . . . . . . . . . . \( \rightarrow S(Lt(py, s(py))) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(Lt(py, s(py))) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(N(s(py)) & Lt(py, s(py))) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(N(py)) & N(s(py)) & Lt(py, s(py))) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(R(py, s(py)) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(R(py, s(py))), S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . (as in subterm induction example)

. . . . . . . . . . . . . \( S(Add(py, 0, py)) \rightarrow S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . \( S(Add(py, 0, py)), S(N(py)) \rightarrow S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . \( S(N(py)) \rightarrow S(N(py)), S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . \( \mathcal{C}(py), S(N(py)) \rightarrow S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . \( S(R(py, s(py)) \supset C(py), S(N(py)) \rightarrow S(Add(s(py), 0, s(py))) \)

. . . . . . . . . . . . . \( S(R(py, y)) \supset C(py), S(y = s(py)), S(N(py)) \rightarrow S(Add(y, 0, y)) \)

. . . . . . . . . . . . . \( S(R(py, y)) \supset C(py), S(y = s(py)) & N(py)) \rightarrow S(Add(y, 0, y)) \)

\( \mathcal{B}(y), S(y = s(py) & N(py)) \rightarrow S(Add(y, 0, y)) \)

where \( \mathcal{C}(z) \equiv S(N(z)) \supset S(Add(z, 0, z)) \) and \( \mathcal{B}(z) \equiv \forall z(S(R(x, z)) \supset C(z)) \)

Figure A-12: Derivation of \( \mathcal{B}(y), S(y = s(py) & N(py)) \rightarrow S(Add(y, 0, y)) \) in LKE+PAR+WF.
## Index of Definitions

<table>
<thead>
<tr>
<th>Term Being Defined</th>
<th>Defn #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antecedent</td>
<td>3.1.1</td>
<td>54</td>
</tr>
<tr>
<td>Assertion</td>
<td>3.1.1</td>
<td>54</td>
</tr>
<tr>
<td>Closure</td>
<td>2.2.1</td>
<td>24</td>
</tr>
<tr>
<td>Existential Closure</td>
<td>1.4.5</td>
<td>18</td>
</tr>
<tr>
<td>Computation sequence:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential-And</td>
<td>2.3.1</td>
<td>32</td>
</tr>
<tr>
<td>Sequential-Or</td>
<td>2.3.1</td>
<td>32</td>
</tr>
<tr>
<td>Sequential-Both</td>
<td>2.3.1</td>
<td>32</td>
</tr>
<tr>
<td>Conclusion</td>
<td>3.2</td>
<td>56</td>
</tr>
<tr>
<td>Consequent</td>
<td>3.1.1</td>
<td>54</td>
</tr>
<tr>
<td>Context:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative</td>
<td>3.4.9</td>
<td>73</td>
</tr>
<tr>
<td>Positive</td>
<td>3.4.9</td>
<td>73</td>
</tr>
<tr>
<td>Divergence</td>
<td>2.2.2</td>
<td>27</td>
</tr>
<tr>
<td>Existential Closure</td>
<td>1.4.5</td>
<td>18</td>
</tr>
<tr>
<td>Failure</td>
<td>2.2.2</td>
<td>27</td>
</tr>
<tr>
<td>First Order Language</td>
<td>1.4.1</td>
<td>17</td>
</tr>
<tr>
<td>Formula:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td>1.4.2</td>
<td>17</td>
</tr>
<tr>
<td>Signed</td>
<td>3.1.1</td>
<td>54</td>
</tr>
<tr>
<td>Guaranteed Termination</td>
<td>5.1.2</td>
<td>116</td>
</tr>
<tr>
<td>$IH(S,f,x)$</td>
<td>5.3.1</td>
<td>125</td>
</tr>
<tr>
<td>Key Subformula</td>
<td>4.2.1</td>
<td>88</td>
</tr>
<tr>
<td>Predicate Definition</td>
<td>1.4.3</td>
<td>18</td>
</tr>
<tr>
<td>Premiss</td>
<td>3.2</td>
<td>56</td>
</tr>
<tr>
<td>Program</td>
<td>1.4.4</td>
<td>18</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>3.5.3</td>
<td>77</td>
</tr>
<tr>
<td>Program-Formula Judgement</td>
<td>5.1.6</td>
<td>118</td>
</tr>
<tr>
<td>Topic</td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>Renaming-Permutation Equivalence</td>
<td>2.2.3</td>
<td>27</td>
</tr>
<tr>
<td>Sequent</td>
<td>3.1.1</td>
<td>54</td>
</tr>
<tr>
<td>Stack:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backtrack</td>
<td>2.2.1</td>
<td>24</td>
</tr>
<tr>
<td>Goal</td>
<td>2.2.1</td>
<td>24</td>
</tr>
<tr>
<td>Substitution:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>More specific</td>
<td>2.5.3</td>
<td>47</td>
</tr>
<tr>
<td>More general</td>
<td>2.5.3</td>
<td>47</td>
</tr>
<tr>
<td>Success</td>
<td>2.2.2</td>
<td>27</td>
</tr>
<tr>
<td>Term</td>
<td>1.4.1</td>
<td>17</td>
</tr>
<tr>
<td>Unfolding:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disjunctive</td>
<td>4.2.1</td>
<td>88</td>
</tr>
<tr>
<td>Predicate</td>
<td>3.3.1</td>
<td>62</td>
</tr>
<tr>
<td>Validity:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Of a closed assertion</td>
<td>3.1.2</td>
<td>55</td>
</tr>
<tr>
<td>Of a sequent</td>
<td>3.1.3</td>
<td>56</td>
</tr>
<tr>
<td>Well-Ordering</td>
<td>5.3.3</td>
<td>128</td>
</tr>
</tbody>
</table>