META LOGICS FOR LOGIC PROGRAMMING

MARIANNE KALSBEEK

Meta-Logics for Logic Programming

ILLC Dissertation Series 1995-13

institute for logic, language and computation

Meta-Logics for Logic Programming

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr P.W.M. de Meijer
ten overstaan van een door het college van dekanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op vrijdag 22 september 1995 te 15.00 uur

door

Marianne Baukje Kalsbeek

geboren te Utrecht

Promotores: Prof. dr J.F.A.K. van Benthem, Prof. dr K.R. Apt
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam

The investigations were supported by the Netherlands Organization for Scientific Research (NWO).

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Kalsbeek, Marianne

Meta-logics for logic programming / Marianne Kalsbeek. -
Amsterdam : Institute for Logic, Language and
Computation, Universiteit van Amsterdam. - Ill. - (ILLC
dissertation series ; 1995-13)
Thesis Universiteit van Amsterdam. - With ref.
ISBN 90-74795-37-4
NUGI 811
Subject headings: logic programming.

Cover design by Ans Gevers
Cover illustration in charcoal by Marianne Kalsbeek
Printed and bound by Haveka B.V., Alblasserdam

voor mijn zusje, Emily

Contents

Acknowledgments

I am very grateful to have had the opportunity to work and study in an extraordinary and pleasant entourage.

First of all, I want to thank the tandem of my two supervisors, Johan van Benthem and Krzysztof Apt, for their encouragement and support. To avoid getting caught between the grinding wheels of their opposite but equally strong characters, I had no choice but to develop a style of my own and make up my own mind most of the time. I enjoyed their sound advice, their guidance, their enthusiasm, and their vast knowledge of relevant literature. I am deeply indebted to Johan, who gave me a chance when all odds where against me.

Several of my colleagues and friends have had a great influence on me these past years. My love for mathematics was first kindled by Michiel van Lambalgen. Without him, it is unlikely that I would ever have turned to mathematics and stuck to it. It was Frank van Harmelen who initiated me in meta-logic programming, with great patience and undying enthousiasm. His attitude was probably the best example I could have had. I owe a lot to the Compulog II community. I want to thank Bob Kowalski, John Lloyd, Pat Hill, Bern Martens, and many others for inspiring discussions and very pleasant gatherings. Salvatore Ruggieri fortunately prevented my including a set of basically flawed results in this thesis. The months of co-authoring the paper on Ambivalent Logic, which has become the first chapter of this thesis, count among the most happy of the past years. I want to thank Jiang for his generosity and his cheerful company. I also want to thank Dov Gabbay, Grisha Mints, and István Németi for discussions and suggestions.

I would like to thank my daily colleagues in Amsterdam for their help and suggestions; I enjoyed Bas Terwijn's daily dose of jokes. The participants in the Pressure Cooker welcomed me as a foreign ingredient in the algebraic stew — any sign of my understanding of what was going on should probably be interpreted as illusory, but I certainly enjoyed being there.

During an extended period I depended heavily on a group of family and friends for very basic daily needs. I am very grateful to my mother, my sister Emily, Ellen Blom, again Michiel, and Hélène van der Knaap, for their help, their company, and their understanding.

I want to thank Yde Venema, Marianne Selie, and Jacques Dob for their company — during the final stage of writing, Jacques provided me with an excellent survival-blend of tea, cigarettes, and poetry. For various reasons, I want to thank Ikumi Imani, Arno Nollen, András Simon, and Ron Neuteboom.

My parents' encouragement, support, and belief in my talents, has meant very much to me. Recently, I started to realise how much I have learned from my father, who unfortunately is not here any more to see where I have arrived. My mother's vitality and her potential for compassion continue to be an inspiring example.

Amsterdam, August 1995

Introduction

This dissertation consists of three separate parts, each of which can be read independently. In each part, a theme in Logic Programming (or rather, the practice of Logic Programming) is taken up and investigated from a logical point of view.

Part I is inspired by the observation that, although Logic Programming is based on first order predicate logic, in many applications and implementations, such as Prolog, meta-logic programming, and databases, a syntax is employed that stretches far beyond first order predicate logic. Ambivalent Logic, essentially first order predicate logic with a very liberal syntax, is developed in Part I; a series of formal results justify the current Logic Programming practice of using liberal versions of first order predicate logic syntax.

In Part II, the focus is on Vanilla meta-programming, where the meta- program takes object programs as input and imitates their execution. Typically, ambivalent syntax is employed here. Various correctness proofs for the standard Vanilla meta-interpreter are discussed and compared.

Part III is based on the observation that in implementations of Logic Programming, the added control affects the procedural meaning of programs. In particular, the standard top-down processing of program clauses induces substructural effects. Substructural (Gentzen style) sequent calculi corresponding to various implementation styles, among them standard Prolog, are investigated here.

Part I

Ambivalent Logic

Chapter 1
A Vademecum of Ambivalent Logic

Ambivalent Logic AL, first introduced in its full sense in Jiang [Jia94a] is obtained from first order predicate logic FOL by relaxing several restrictions on its usual syntax. In particular, the usual distinctions between predicates, functions, formulas and terms are not made in AL. We show that Ambivalent Logic provides a general and flexible framework for various ambivalent syntactic phenomena that occur in Prolog, in meta-logic programming and in formalisations of knowledge and belief. A series of formal results justifies the use of syntaxes with ambivalent phenomena in these areas. We discuss a closed term semantics for AL, and show that the standard derivational calculus for first order predicate logic is sound and complete w.r.t. this semantics. A conservativity result shows that AL should be considered as a conservative extension of FOL. We define a version of the Martelli Montanari unification algorithm for AL, and show it has the usual properties. In combination with various other basic proof theoretic results, this shows that resolution is a complete and sound inference method for AL. The results all relativise to subsystems of AL, like Prolog's syntax. We also discuss the relation with Hilog.

A (mostly tacit) assumption in the syntax for first order predicate logic (FOL) is that the sets of predicates, functions, and constants have mutually empty intersections. As a consequence, the syntactic categories of formulas and terms are mutually exclusive, as the Unique Reading Lemma for FOL witnesses. While Logic Programming is in principle based on FOL, in various of its application areas, syntaxes are used which do not satisfy this property of the usual syntax for FOL.

A principal example, analysed in Apt and Teusink [AT95], is the syntax underlying Prolog. Prolog's meta-variable facility allows for variables to occur both in terms and in atoms positions. Two well-known examples are the cut-fail definition of negation: $neg(X) :- X, !, fail$, and the definition of the solve-predicate as $solve(X) :- X$. Clearly, this use of variables is not allowed in the standard syntax of FOL. Additionally, queries which involve instantiations of the heads of the above clauses with terms $p(t)$, presuppose a syntax which allows for function symbols to be accepted as predicate symbols. Conversely, an inductive definition of a predicate involving its negation presupposes that Prolog's syntax allows for predicate symbols to be accepted as function symbols. Also, in Prolog's syntax, predicates and functions do not have fixed arities.

Another example of the use of a deviant syntax in (meta-)Logic Programming practice is the untyped, non-ground Vanilla meta-interpreter which uses identity naming for atoms and terms of the object level language (cf. Chapter 2). While in this simple, unamalgamated case the (partial) correctness of Vanilla could still be guaranteed by the possibility of renaming the object level predicates to meta-level function symbols, this solution does not generalise to various interesting extensions of Vanilla. For example, extending Vanilla with reflective clauses like $demo(X) :- X$ or instances of these clauses, eliminates the possibility of renaming and introduces ambivalence between function and predicate symbols and variables and terms, similar to the ambivalences observed above in the case of Prolog's syntax. A related example is the formalisation of the Three Wise Man problem as an extension of Vanilla in Kowalski and Kim [KK91]. This formalisation involves clauses of the form $demo(wise0, not\ demo(wise1, white1)) :-$, in which the predicates *demo* and *not* occur in function positions.

The above examples require only limited forms of ambivalence: atoms-as-terms ambivalence (atoms occurring in term positions) and terms-as-atoms ambivalence (terms occurring in atom positions). As observed in Chen et al. [CKW93] and Jiang [Jia94a], more advanced ambivalent features are required to obtain a syntax which allows for efficient formalisations of generic predicates. An example is the generic closure predicate $(cl(Z))(X,Y)$, which, given any binary predicate R, returns its transitive closure $cl(R)$. Its definition, $((cl(Z))(X,Y) \leftrightarrow Z(X,Y) \vee (Z(X,V) \wedge (cl(Z))(V,Y))$, presupposes a syntax which allows for predicates to occur in term positions, and for variables and atoms to occur in predicate positions. Similar ambivalent phenomena occur in languages which allow for data retrieval and schema browsing. In databases, such forms of syntactic ambivalence are a desirable option, allowing caching of data.

Other areas where ambivalent phenomena occur, are formalisations of knowledge and belief. While the use of a real naming function, avoiding syntactic ambivalence between terms and formulas, is often possible, it is not always desirable. Also, the above atoms-terms ambivalence is not always sufficient. In the predicate case, it is desirable that a reflective axiom like $x \rightarrow K(x)$ can be instantiated with all formulas, not just atomic formulas. This then supposes a syntax in which all formulas, and not just atomic formulas, are allowed to occur as terms. While the quantifiers could, in principle, be represented by functions, the effect of a functional representation does not in all cases give the desired effect. The obvious reason is that quantifiers, in contrast to functions, bind variables. For example, consider the formula $\forall x.bel(John, (friend(John, x) \rightarrow exists(y, loves(x, y)))$, which expresses the proposition 'John believes about all his friends that they are loved by somebody'. Instantiation of both x and y with the same constant is possible, yielding unintended statements like

$$bel(John, (friend(John, Mary) \rightarrow exists(Mary, loves(Mary, Mary))).$$

In contrast, the use of a quantifier representation for the same proposition yields the formula $\forall x.bel(John, (friend(John, x) \rightarrow \exists y.loves(x, y)))$. Here y is bound by the existential quantifier, and cannot be instantiated. Thus, unintended instantiations like the above are not possible.

In the present paper, we discuss Ambivalent Logic AL as a general framework for first order logic with a syntax in which all of the above mentioned ambivalent phenomena occur. AL has a fully ambivalent syntax, in which the usual distinctions between the syntactic categories of terms, functions, predicates, and formulas, cannot be made. While the part of Unique Reading that says that a well-formed string in the language is either a formula or a term, but not both, does not hold for AL, these syntactic distinctions do retain their usual contextual meaning.

AL has a standard (first order predicate logic) derivational calculus, and a (first order) closed term semantics. We develop some basic proof theory for AL, including soundness and completeness of the derivational calculus w.r.t. the semantics, an s-equivalence theorem, and Herbrand's theorem. We discuss unification for AL. In particular, we show how the Martelli-Montanari unification algorithm can be adapted for AL. We prove that the appropriate AL version has the usual properties. In combination with various other results discussed in this paper, this shows that resolution is a complete and sound inference method for AL. In addition, we show that AL is a conservative extension of FOL. We also show how various known ambivalent syntaxes (such as Prolog's syntax and the syntax discussed in Kalsbeek [Kal95a] and Chapter 2) can be obtained as special instances of the full ambivalent syntax of AL. The proof theoretic results we obtain for AL also hold for the various specialisations of AL. These results justify the use of AL and various of its subsystems as a basis for (meta-) logic programming. In addition, we argue that AL provides an interesting framework for formalisations of knowledge and belief.

AL was first introduced in Jiang [Jia94a] and [Jia94b]. Some of the results we present are improvements of results announced in Jiang [Jia94a]. A proper subsystem of AL, appropriate for amalgamated extensions of the Vanilla meta-interpreter, was introduced in Kalsbeek [Kal93], and will be discussed in some detail in Chapter 2.

There are various examples of logics which, like AL, have a syntax incorporating ambivalent phenomena. Recently, Hilog was proposed by Chen et al. [CKW93] as a basis for higher order logic programming. Both AL and Hilog combine a second order syntax with a first order semantics. While the syntax of AL extends Hilog syntax, Hilog is, in contrast to AL, not an extension of FOL. We will investigate the differences between AL and Hilog in Section 1.6. Another example is the logic proposed by Richards [Ric74], which is intended for formalisations of intensional logics. For a comparison between AL and Richards' logic, we refer to Jiang [Jia94a]. The reader is also referred to Gabbay [Gab92], where other flexible meta-languages are proposed.

Outline

In Section 1.1, we introduce the fully ambivalent syntax of AL and we show how specialisations of it may be obtained. In Section 1.2 we develop a closed term semantics for AL. In Section 1.3 we discuss how equality can be incorporated in AL. Section 1.4 is devoted to various basic proof theoretic results for AL. In Section 1.5 we discuss the appropriate version of the Martelli-Montanari unification algorithm for AL. In Section 1.6 we discuss the relations between Ambivalent Logic and Hilog.

1.1 Syntax for Ambivalent Logic

We define in Section 1.1.1, a syntax which is fully ambivalent, that is, in which every well formed expression can act as a formula, as a term, as a function, and as a predicate. Whether an expression is evaluated as a term, a formula, a function, or a predicate will be determined by the context. We allow for free arity of predicates and functions. This syntax generates a multi-purpose language.

The full ambivalent syntax extends the syntax for the Vanilla meta-interpreter (cf. Chapter 2), in which can occur in term positions but not vice versa, and in which predicates and functions are always symbols with a fixed arity. It also extends Prolog's syntax (cf. Apt and Teusink [AT95]), which shares with full ambivalent syntax the full atoms-terms ambivalence, and the free arity of functions and predicates, but in which predicates and functions are always parameters.

To obtain versions of ambivalent syntax which generate languages that are adapted to a particular purposes such as the above, the definitions for full ambivalent syntax can be adapted and specialised. In Section 1.1.2 we discuss several of these refinements.

1.1.1 Full ambivalent syntax

A fully ambivalent language $L_{\mathcal{G}}$ is generated by a set of non-logical constants (parameters) $\mathcal{G}$, an infinite set of variables $x, y, z, \ldots$, and the usual logical connectives and quantifiers of first-order logic.

1.1.1. Definition. (Expressions)

1. The variables $x, y, z, \ldots$ and individual constants $a, b, c, \ldots \in \mathcal{G}$ are expressions.
2. If t, $t_1, \ldots, t_n$ are expressions, then $(t)(t_1, \ldots, t_n)$ is an expression.
3. If A and B are expressions, then so are $\neg A$, $A \wedge B$, $A \vee B$, $A \leftarrow B$, $A \rightarrow B$ and $A \leftrightarrow B$.
4. If A is an expression and x is a variable, then $\forall x(A)$ and $\exists x(A)$ are expressions. $\square$

The above ambivalent syntax thus extends standard syntax for first order predicate logic in several ways. The usual requirement is dropped which states that the sets of predicate symbols and function symbols are disjoint. In addition, the usual requirement is dropped that predicates and functions have a fixed arity. As an example, $p(p(p, p))$ is a well-formed ambivalent expression. In addition, variables may occur in formula positions. As an example, the Prolog clause $solve(x) \leftarrow x$ is a well-formed expression in full ambivalent syntax. Also, 'second order' quantification is allowed. That is, expressions like $\exists x.x(c)$ are well-formed in full ambivalent syntax. (We will see in Section 1.2 that, semantically, quantification over predicates will not be interpreted as second order quantification.) Moreover, not only parameters, but also more complex expressions are allowed as predicates and functions. An example is the generic closure predicate discussed in the introduction. This feature allows for the formation of new predicates, which can be useful in databases. Other examples are: $p(x) \wedge q(x) \rightarrow (p \wedge q)(x)$ and $\forall x \forall z(p(x, z) \rightarrow (\exists y.p(y))(x))$. The latter example shows that quantified expressions are allowed in predicate places.

In many cases, we will omit brackets if this does not lead to confusion. For example, we will write $c(x)$ instead of $(c)(x)$, and $\forall xp(x)$ (or also $\forall x.p(x)$) instead of $\forall x(p(x))$. In various cases, however, brackets cannot be omitted without altering an expression. For instance, $(p \vee q)(t, s)$ is an atomic expression, while $p \vee q(t, s)$ is a disjunctive expression. Similarly, we distinguish between the expressions $p((x)(a))$ and $(p(x))(a)$. In the former, the symbol p predicates over the expression (x)(a), while in the latter, $p(x)$ predicates over a.

1.1.2. Definition. (Atomic expressions)
An *atomic expression* is either a constant, or a variable, or an expression of the form $(t)(t_1, \ldots, t_n)$.
Atomic expressions of the form $(t)(t_1, \ldots, t_n)$ are *functional atoms.* $\square$

The set $FV(t)$ of free variables of an expression t is defined analogous to the definition for standard syntax, except that it is additionally defined for expressions in predicate and function places.

1.1.3. Definition. (Free Variables) Let x be a variable, $c \in \mathcal{G}$, and let A, B, and $(t)(t_1, \ldots, t_n)$ be expressions. The set of free variables occurring in an expression is defined as follows:

$$FV(x) = x$$
$$FV(c) = \emptyset$$
$$FV((t)(t_1, \ldots, t_n)) = FV(t) \cup FV(t_1) \cup .. \cup FV(t_n)$$

$FV(A \wedge B) = FV(A) \cup FV(B)$
$FV(\neg A) = FV(A)$
$FV(\exists x A) = FV(A) \setminus \{x\}$
Similarly for the other binary connectives and the universal quantifier.

A variable x occurring in an expression t is *bound* in t if $x \notin FV(t)$; otherwise we say x occurs free in t. We write $A\{x/t\}$ to denote the result of replacing every free occurrence of x in A by t. □

1.1.4. DEFINITION. An expression t is *closed* (or a *sentence*) if $FV(t) = \emptyset$.

By Definition 1.1.3, x occurs free in $(p(x))(a)$. Also, x is not free in $p(\exists x p(x))$ because x is not free in the argument $\exists x p(x)$. In this sense, quantifiers in term-positions behave like ordinary quantifiers.

The above definition can be refined in a standard way to distinguish between various occurrences. For example, in $p(x) \vee \exists x f(x)$, the first occurrence of x is free, while the second occurrence is bound by the existential quantifier. The scope of quantifiers can be defined in the usual way. For example, in $\exists x q(p(x) \wedge \exists x f(x))$, the first occurrence of x is bound by the outermost existential quantifier, while the second occurrence is bound by the rightmost quantifier.

In Section 1.2 we will need the following standard notion.

1.1.5. DEFINITION. *An expression t is* free for *a variable x in an expression A iff t does not contain any free variable that is bound by some quantifier in A when every free occurrence of x in A is replaced by t.* □

Due to the nature of full ambivalent syntax, the role of an expression can be determined only by the context of the expression. For example, consider the expression $(q(d))(a, q(a)) \wedge q(c)$. It can be evaluated both as a term and as formula, depending on the context. In the latter case, q(d) serves as a predicate, q(a) as a term, and q(c) as a formula.

Clearly, the Unique Reading Lemma does not hold for ambivalent syntax, as every AL expression can be both a formula and a term. We can, however, define in some cases which role a subexpression assumes in the context of an expression in which it appears. In some cases, it is trivial to determine the contextual role of subexpressions. For example, $a \vee b$ occurs as a term in $a(a \vee b)$, independent of whether the latter is evaluated as a term or as a formula. But we have the choice whether or not to consider a as a term in $c(a \vee b)$. The choice we make is inspired by the semantics we define in the next section. In this semantics, there will be no connection between the interpretation of the 'term' a and the interpretation of the 'term' $a \vee b$. Therefore, we will not consider a as a subterm in $c(a \vee b)$. Similar considerations lead to the following definitions of the notions of subformula, term, function, and predicate. The definition of the notion of subformula is standard.

1.1.6. DEFINITION. (Subformula)

1. Every expression is a subformula of itself.

2. Let A be an expression. Then every subformula of A is a subformula of $\neg A$, $\exists x A$, and $\forall x A$.
3. Let A and B be expressions. If E is a subformula of A or B, then E is a subformula of $A \vee B$, $A \wedge B$, $A \rightarrow B$, $A \leftarrow B$, and $A \leftrightarrow B$. □

By this definition, a does not occur as a subformula in $c(a \vee b)$.

1.1.7. DEFINITION. (Occurrence as a term)

1. In an expression of the form $(t)(t_1, \ldots, t_n)$, $t_1, \ldots, t_n$ occur as terms.
2. If t occurs as a term in A and A is a subformula of F, then t occurs as a term in F.
3. If t occurs as a term in s and s occurs as a term in F, then t occurs as a term in F. □

By the above definition, a does not occur as a term in $c(a \vee b)$, while $a \vee b$ does. Also, c does not occur as a term in $c(a \vee b)$.

1.1.8. DEFINITION. (Occurrence as a function)

1. t occurs as a function in an expression $(t)(t_1, \ldots, t_n)$.
2. If t occurs as a function in s and s occurs as a term in F, then t occurs as a function in F. □

1.1.9. DEFINITION. (Occurrence as a predicate)

1. t occurs as a predicate in an expression $(t)(t_1, \ldots, t_n)$.
2. If t occurs as a predicate in A and A is a subformula of F, then t occurs as a predicate in F. □

By these definitions, t occurs both as a predicate and as a function in $(t)(t_1, \ldots, t_n)$. In $\forall x(t(x))$, t occurs as a predicate, but not as a function. In $p(\forall x(t(x)))$, t occurs neither as a predicate nor as a function.

All of the above definitions can be refined to distinguish between the various occurrences. For example, in $a(a) \vee a$, the first occurrence of a is as a predicate (but not as a function), the second occurrence is as a term, while the third occurrence is as a subformula.

It is useful to make the following distinction between two kinds of occurrences of quantifiers. The first kind of quantifier, which we will call 'outside quantifier', occurs in places where they are also allowed in FOL formulas. The second kind occurs only in 'ambivalent' places.

1.1.10. DEFINITION. Let $Qx.s$, where Q is a quantifier $\forall$ or $\exists$, be an occurrence of a subexpression of an AL expression t. Q is an *outside quantifier* if it is an occurrence as a subformula of t. Otherwise, Q is an *inside quantifier*.
Inside and outside connectives are defined similarly. □

For example, in $\forall x.(x \vee \exists y.f(y))$, the first quantifier is an outside quantifier, while the second is an inside quantifier. Observe that both inside and outside quantifiers do bind variables in their scope. In Section 1.2, where we discuss semantics for AL, we will see that the outside quantifiers will be interpreted as real quantifiers, ranging over elements of the domain; this in contrast to inside quantifiers.

1.1.2 Refinements

For many purposes, the full ambivalent syntax defined in the previous subsection admits too many expressions. Refined versions of ambivalent syntax, well-tuned to particular domains of application, can be obtained by specialising one or more of the clauses in Definition 1.1.1, and by the use of sets of special constants in addition to the set of generating constants. In defining special versions of full ambivalent syntax, it is sometimes useful to introduce new syntactic categories.

We will give some examples of specialisation and the use of special constant sets.

- A version of ambivalent syntax which does not admit all its expressions to occur as predicates and functions, but only its constants, can be obtained by modifying clause (2) of Definition 1.1.1 as follows: If $t_1, \ldots, t_n$ are expressions and $c \in \mathcal{G}$, then $c(t_1, \ldots, t_n)$ is an expression. In this particular version, variables and more complex expressions do not occur as predicates and functions, and as a result there is no 'second order' quantification.
- It may be useful to select special roles for some of the parameters. As an example, in some domains, it may be useful to have one or more symbols that occur only as predicates. The negation predicate in Prolog is an example of such a special predicate (see below). In that case, a set of special symbols $\mathcal{S}$ is added to the signature of the language, and an extra syntactic category is introduced: special expressions. The following modification of Definition 1.1.1 is used:

  1 Variables x and parameters $c \in \mathcal{G}$ are expressions.
  2 If t, $t_1, \ldots, t_n$ are expressions, then $(t)(t_1, \ldots, t_n)$ is an expression.
  2' If $p \in \mathcal{S}$, and $t_1, \ldots, t_n$ are expressions, then $p(t_1, \ldots, t_n)$ is a special expression.
  3' If A is an expression, then $\neg A$ is an expression; If A is a special expression, then $\neg A$ is a special expression.
  $A \vee B$ is a special expression if both A and B are special expressions or if one among them is a special expression and the other is an expression.
  $A \vee B$ is an expression if both A and B are expressions.
  Similarly for the other binary connectives.
  4' $\forall x(A)$ is an expression if A is an expression; it is a special expression if A is a special expression.

  In particular, special expressions do not occur as terms, predicates, or functions, in expressions and special expressions. Special parameters only occur as predicates in special expressions.
  Special definitional clauses can also introduce parameter symbols which only occur with fixed arities.
- The syntax of Prolog is a specialisation of full ambivalent syntax, and shares some of the properties of the above specialisations. In Prolog's syntax, only parameters are allowed in predicate and function positions. In addition, Prolog has a special predicate *not*, which is only allowed to occur in predicate positions. A representative part of Prolog's syntax can be described as follows:

1 variables x and parameters $c \in G$ are expressions.
2 $c(t_1, \dots, t_n)$ is an expression if $c \in G$ and $t_1, \dots, t_n$ are expressions.
3 $not(t)$ is a special expression if t is an expression.
4 $fail$ and ! are special expressions.
5 every expression is a special expression.
6 $t \leftarrow t_1, \dots, t_n$ is a Prolog clause if t is an expression and $t_1, \dots, t_n$ are expressions.
7 every expression that is not a variable is a Prolog clause.

Other features of Prolog's syntax can be incorporated in this framework, such as the Prolog built in predicates *assert* and *retract*, which take Prolog clauses as arguments.

- The ambivalent syntax described in Kalsbeek [Kal95a] and Chapter 2 is a specialised version of full ambivalent syntax. It differs from full ambivalent syntax in the following ways:
  1. Only atomic expressions are allowed to occur in term positions.
  2. Only parameters are allowed to occur as functions and predicates.

  In particular, variables are not allowed to occur in formula positions. A full description can be found in Chapter 2.

1.2 Semantics

We have chosen to develop a closed term semantics for AL for the purpose of this paper. All of the definitions we give can be adapted to special versions of AL such as Prolog's syntax.

Formally, a structure $\mathcal{M}$ for an ambivalent language L_G (the *underlying language* of $\mathcal{M}$) is a tuple (D, T), where

1. D, the domain of $\mathcal{M}$, is the set of *closed* expressions of L_G;
2. T, the truth set of $\mathcal{M}$, is a subset of the set of *closed atomic* expressions of L_G.

The satisfiability relation for closed expressions in a structure $\mathcal{M} = (D, T)$ is inductively defined as follows:

1.2.1. DEFINITION.

$\mathcal{M} \models_{AL} A$ iff $A \in T$ where A is a closed atomic expression
$\mathcal{M} \models_{AL} A \wedge B$ iff $\mathcal{M} \models_{AL} A$ and $\mathcal{M} \models_{AL} B$
$\mathcal{M} \models_{AL} \neg A$ iff $\mathcal{M} \not\models_{AL}$ A
$\mathcal{M} \models_{AL} \forall x A$ iff for all $d \in D$, $\mathcal{M} \models_{AL}$ A{x/d}
Disjunction, implications, equivalence and existential quantification are defined in the standard way. □

Several things are worth noting at this stage.

- Closed expressions can be evaluated as sentences in a model. At the same time, they constitute elements of the domain.

- A closed expression, evaluated as a sentence, in a model, has a unique truth value. Thus, while there is syntactic ambivalence, there is no semantic ambiguity.
- The truth values of atoms are, in principle, not related to the structure of expressions which occur in them as subterms. That is, for example, the truth value of an atom $p(s \wedge v)$ is in principle not related to the respective truth values of $p(s)$ and $p(t)$. A similar distinction holds for the relation between outside and inside quantifiers: The truth value of $\forall x.p(f(x))$ is independent of the truth value of $p(\forall x.f(x))$. The latter is decided by a checking whether the atom $p(\forall x.f(x))$ belongs to the truth set, while the former is decided by checking whether $p(f(d))$ belongs to the truth set, for each element d in the domain. In certain applications, such as formalisations of knowledge and belief, it might be desirable to have explicit relations between inside and outside quantifiers, or between outside and inside connectives such as the above. The soundness and strong completeness theorem 1.4.1 that we will prove in Section 1.4 shows that such relations may be implemented by axioms.
- It should be noted that 'similar' expressions like, for example, $\forall x.f(x)$ and $\forall y.f(y)$ constitute different, and unrelated, objects in the domains of models. That is, the truth values of the closed expressions $t(\forall x.f(x))$ and $t(\forall y.f(y))$ need not be the same. This may be counterintuitive, and even undesirable in some applications. Extra assumptions on the truth sets T may impose that expressions which are similar under appropriate renaming of the bound variables, behave similarly as elements of the domain. This in its turn should then be matched with an appropriate rule in the derivational calculus. In contrast, for outside quantifiers the following relation, familiar from FOL, holds: $\mathcal{M} \models \forall x A$ iff $\mathcal{M} \models \forall y A\{x/y\}$, if y is free for x in A.
- It should be observed that in the above interpretation of the quantifiers (the substitution interpretation), there is no real semantic second order quantification: the quantifiers range over object in domains, and not over subsets of domains. Thus, while syntactically AL allows for second order features such as quantification over functions and predicates, its closed term semantics is first order.

We will use the following standard notions.

1.2.2. DEFINITION. Let A be a closed expression and let S be a set of AL sentences. S is *satisfiable* in AL iff there exists an interpretation $\mathcal{M}$ such that $\mathcal{M} \models_{AL} \phi$ for all $\phi \in S$. $\mathcal{M}$ is called a *model* of S in this case.
A is *valid* in AL, denoted by $\models_{AL} A$, iff $\neg A$ is not satisfiable in AL.
We say that A is a *logical consequence* of S, denoted by $S \models_{AL} A$, iff A is true in every model of S. □

1.2.3. DEFINITION. A structure $\mathcal{M}$ is a *Herbrand model* for a theory S ($\mathcal{M} \models^h_{AL} S$) if $\mathcal{M}$ is a structure for S and the underlying language of $\mathcal{M}$ is the underlying language of S (that is, if D coincides with the set of closed expressions of the full ambivalent syntax generated by S). □

In addition, we can define semantics for all expressions, using assignments (Definition 1.2.5). We need the following standard definitions.

1.2.4. DEFINITION. Let σ be a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$. Then its *domain* $dom(\sigma)$ and *range* $ran(\sigma)$ are defined as follows:
$dom(\sigma) = \{x_1, \ldots, x_n\}$,
$ran(\sigma) = \{t_1, \ldots, t_n\}$
The σ-image of a variable x, denoted as $x\sigma$, is given as follows:
$x_i\sigma = t_i$, for $i \in [1, n]$, and
$x\sigma = x$, for $x \notin dom(\sigma)$.
For expressions t and substitutions σ, $t\sigma$ is the result of replacing each free occurrence of x in t by the σ-image of x. □

1.2.5. DEFINITION. A substitution σ is an *assignment* for an expression s and an interpretation $\mathcal{M} = \langle D, T\rangle$, if $dom(\sigma) \supseteq FV(s)$ and $ran(\sigma) \subseteq D$. □

1.2.6. DEFINITION. Let σ be an assignment for t and $\mathcal{M}$. We say that $\mathcal{M}$ *satisfies t with σ* ($\mathcal{M}, \sigma \models t$) if $\mathcal{M} \models t\sigma$. □

As a *derivational calculus* for Ambivalent Logic we take some standard system of natural deduction. The resulting derivability relation will, as usual, be indicated by $\vdash$. The deduction rules are defined in a standard way. Consider the usual elimination rule for the universal quantifier, $\forall x.t/t\{x/s\}$. It has a side condition, which prevents unintended bindings resulting from the substitution of s for x. In the case of AL, unintended bindings can result, not only from quantifiers in standard places, but also from inside quantifiers. As an example, $\forall x p(\exists y f(x,y))/p(\exists y f(y,y))$ is not a correct instance of the rule for elimination of the universal quantifier: y is not free for x in $p(\exists y f(x,y))$.

1.2.7. EXAMPLE. The ambivalent expression $p(a)$ can, among others, be considered as either $p(x)\{x/a\}$ or $x(a)\{x/p\}$. The former corresponds to the following (standard) instance of the introduction rule for the existential quantifier: $p(a)/\exists x(p(x))$. The latter in its turn corresponds to $p(a)/\exists x(x(a))$.

We remind the reader of the following two notions of completeness.

1.2.8. DEFINITION. Let L be a logic with semantic consequence relation $\models_L$ and derivational consequence relation $\vdash_L$.
L is *weakly complete* if, for any sentence B, $\models_L B$ implies $\vdash_L B$.
L strongly complete if, for any sentence B and any set of sentences S, $S \models_L B$ implies $S \vdash_L B$. □

In the case of first order logic with standard syntax, a restriction to Herbrand semantics results in the weakening of some of the standard results. A well-known example is the loss of strong completeness in its full sense. While strong completeness w.r.t. Herbrand models holds for finite and infinitely extendible theories, it does not hold for theories in which every closed term of the underlying language occurs in the theory. The same phenomenon occurs in the context of Ambivalent Logic, as witnessed by the following proposition.

1.2.9. PROPOSITION. *AL is not strongly complete for infinite theories with respect to the Herbrand semantics.*

Proof: Let $t_1, t_2, t_3, \ldots$ be an enumeration of all the closed expressions of an ambivalent language L.
Then, by the definition of validity in Herbrand models, $t_1, t_2, t_3, \ldots \models^h_{AL} \forall x.x$. However, $t_1, t_2, t_3, \ldots \not\vdash \forall x.x$ □

While a restriction to Herbrand semantics is sensible in the case of logic programming (logic programs being finite theories), in a broader context it is, in view of the above proposition, sensible to consider the more general closed term semantics as the appropriate semantics for Ambivalent Logic.

A more general semantics for AL, with arbitrary domains and, consequently, non-identity interpretation functions, will not be considered in the context of the present paper. The reason is that, (as remarked in Chen et al. [CKW93]) interpretation of quantified terms using a non-identity interpretation function requires something like lambda-abstraction, which considerably complicates the construction of models. In the next section we argue that the restriction to closed term semantics is not a severe limitation.

1.3 Equality and Identity

The usual semantics for FOL with standard syntax is different from the closed term semantics we described above. In particular, in the semantics for FOL, any set can in principle serve as the domain of a model, and the interpretation functions, mapping closed terms of the language on elements of the domain, are not necessarily identity functions. In this semantics, equality is usually interpreted as identity on the domains, although the equality axioms in themselves do not force this interpretation. In contrast, closed term models are not appropriate for the identity interpretation of equality.

Closed term models, however, can be used to represent other models. Hence they play an important, albeit hidden, role in FOL. Consider the usual proof of the completeness theorem for FOL, using the Henkin method. The proof in fact consists of two separate stages, each the proof of an independent lemma. In the first stage, a consistent theory T is extended to a maximally consistent Henkin theory T^*, for which a Herbrand model H is constructed. H is a closed term model for the theory T, and, in presence of equality in the language, equality is interpreted in H as an equivalence relation which is also a congruence with respect to the predicates and functions. The second stage of the proof is merely motivated by the convention to interpret equality as identity. From H, a model M for T^* is constructed, the domain of which consists of the equivalence classes in H under equality. The valuation function of M is 'inherited' from H. This second stage can be interpreted as the proof of a representation lemma, which expresses that for every closed term model H there exists an elementary equivalent model M in which equality is interpreted as identity. This representation lemma can be reversed. Let $M = \langle D_M, V_M \rangle$ be a model. A closed term model H is now constructed as follows. Define a function f from the closed terms of the language to be interpreted to the domain of M, which is defined by $f(t) := t^M$. Now define the valuation function V_H as follows: $V_H(P(\bar{t})) := V_M(P(\overline{f(t)}))$. Now H and M are elementary equivalent, and equality is interpreted as just another binary predicate in H, satisfying properties dictated by the equality axioms.

The two sides of this representation result together in fact show that there is a bijection between the class of closed term models and the class of general models (modulo isomorphism). It also shows that there is no inherent need to interpret identity as equality. The interpretation of equality as identity forces the choice of the usual semantics of FOL — if this restriction is abandoned, closed term models are a sound and complete semantics for first order logic with equality, provided the truth sets satisfy the properties dictated by the equality axioms. What we have argued here are several things. First, the usual axioms for equality do in themselves not force the identity interpretation. Second, if the usual identity interpretation of equality is abandoned, we can restrict ourselves to consideration of closed term models without loss of generality.

In particular, we can incorporate equality in Ambivalent Logic without changing the style of the semantics for Ambivalent Logic. In the closed term semantics, equality will not correspond to real identity on the domains.

Let us consider, in some more detail, the usual equality theory ET for FOL. It can be axiomatised (abstracting from arities) as follows:

$$\begin{array}{ll}
\text{(I)} & \forall x.x = x \\
 & \forall x \forall y (x = y \rightarrow y = x) \\
 & \forall x \forall y \forall z (x = y \rightarrow (y = z \rightarrow x = z)) \\
\text{(II)} & \forall x \forall y \forall t (x = y \rightarrow t = t\{x/y\}) \\
\text{(III)} & \forall x \forall y \forall t (x = y \rightarrow (t \rightarrow t\{x/y\}))
\end{array}$$

We obtain an appropriate closed term semantics for AL extended with these equality axioms, by additionally restricting the truth sets to satisfy the corresponding properties. In particular, ET(I) requires truth sets on which $=$ is an equivalence relation. ET(I,II) requires truth sets T (say, with underlying language L_G) which additionally satisfy the following: if $d = c \in T$, and t is an L_G-expression with only x free, then $t\{x/d\} = t\{x/c\} \in T$. ET as a whole requires truth sets T which in addition also satisfies the following property: for all closed L_G-expressions d and c, and for all L_G-expressions t with only x free, if $d = c \in T$, then $t\{x/d\} \in T$ iff $t\{x/c\} \in T$.

Observe that, in the context of the closed term semantics style, we are not committed to ET as a whole. In particular, there are interesting differences between AL + ET(I,II) and AL + ET, which we will more closely consider in the next section.

In the sequel, when we consider AL, we explicitly mean AL without equality. If we consider either of AL + ET(I), AL + ET(II), and AL + ET, we will implicitly assume that the semantics satisfies the corresponding conditions mentioned above.

1.4 Formal results

In the present section, we set out to develop some basic theory for Ambivalent Logic. First, we prove soundness and completeness of AL w.r.t. the closed term semantics.

1.4.1. THEOREM.
AL is sound and strongly complete with respect to the closed term semantics;
AL is strongly complete for infinitely extendible theories with respect to Herbrand semantics.

Proof: Soundness is left to the reader. The completeness proof follows the standard completeness proof for FOL.

Let S be a consistent AL theory with underlying language $L = L_{\mathcal{G}}$. First extend S to a Henkin theory S^*, as follows.
Extend the set of generating constants $\mathcal{G}$ with a new constant d, and let $L' = L_{\mathcal{G}\cup\{d\}}$.

Let $\phi_1, \phi_2, \phi_3, \ldots$ be an enumeration of all the expressions of L' with only x free. Let $d_1, d_2, d_3, \ldots$ be an enumeration of all the closed expressions of L' that are not expressions of L. A carefully chosen subset of these expressions will serve as Henkin constants. Define C_n as the set of closed expressions that occur as terms in $\phi_1 \wedge \ldots \wedge \phi_n$.

Define

$$\begin{array}{lll} S_0 & := & S, \\ m_1 & := & min\{k | d_k \notin C_1\}, \\ S_1 & := & S_0 \cup \{\exists x.\phi_1(x) \rightarrow \phi_1(d_{m_1}), \\ m_{n+1} & := & min\{k > m_n | d_k \notin C_{n+1}\}, \\ S_{n+1} & := & S_n \cup \{\exists x.\phi_{n+1}(x) \rightarrow \phi_{n+1}(d_{m_{n+1}}), \\ S^* & := & \cup S_n. \end{array}$$

S^* is a Henkin theory. Conservativity of S_{n+1} over S_n can be proven in the usual way. Therefore, S^* is conservative over S and thus consistent.

Next, extend S^* to a maximally consistent theory S^m, by the Lindenbaum lemma. The language of S^m is L', and S^m is a Henkin theory. A model $\mathcal{M} = \langle D, T\rangle$ for S^m is obtained as follows. Let D consist of all the closed expressions of L'. Let T consist of the closed atomic expressions that belong to S^m. By induction on the complexity of sentences, $\mathcal{M}$ is a Herbrand model for S^m. By conservativity of S^m over S, $\mathcal{M}$ is a closed term model for S.

For infinitely extendible theories a model can be constructed in a slightly more elegant way. The closed expressions of L that do not occur as terms in the theory can be used as the Henkin constants. The resulting model is a Herbrand model for the theory. □

We leave it to the reader to check that the above result also goes through for AL + ET(I), AL + ET(I,II), and AL + ET.

1.4.1. PROPOSITION. *Let S be a finite set of AL expressions, in which at least one parameter occurs. Then S is infinitely extendible.*

Proof: Let g be a parameter occurring in S. Then $g, g(g), g(g(g)), \ldots$ is an infinite set of closed expressions in the language underlying S. In contrast, the number of closed expressions of this language that occur as terms in expressions of S, is finite, as S is a finite set. □

An immediate consequence of the above theorem and proposition is the following.

1.4.2. COROLLARY. *Every satisfiable AL sentence has a Herbrand model.*

The following notion of normal form is the appropriate version for AL.

1.4.3. DEFINITION. An expression F is in *normal form* if $F \equiv Q_1x_1, \dots, Q_nx_n.t$, where

1. The Q_i are quantifiers $\forall$ or $\exists$.
2. t is built up from atomic expressions and the connectives $\neg$, $\vee$, and $\wedge$; in particular, in any subexpression of t of the form $Qy.s$, where Q is a quantifier, Q is an inside quantifier.
3. The variables x_i are mutually distinct.
4. If $Qy.s$ is a subexpression of t, where Q is an (inside) quantifier, then y is distinct from any of the x_i. □

1.4.4. LEMMA. *For every formula F there is an equivalent formula F', such that F' is in normal form.*

Proof: By the usual methods. □

The *Skolem form* F^s of a normal form expression F can be obtained by the usual methods (cf., for instance, Schöning [Sch89] for an algorithm to obtain Skolem forms). Observe that Skolemisation does leave the inside quantifiers and connectives intact.

Next we prove an AL version of the usual s-equivalence theorem. The proof differs from the usual proof in a crucial aspect. In the usual proof, essential use is made of the option to use non-trivial interpretation functions on domains. In particular, a model $\mathcal{M}$ for a normal form formula F is extended to a model for its Skolem form by extending the signature of $\mathcal{M}$ with functions on its domain that interpret witnesses of the existential quantifiers. In the case of Ambivalent Logic, this construction cannot be used, as we do not allow for non-trivial interpretation functions. For clarity of this exposition, assume that $\mathcal{M}$ is a Herbrand model for F. The introduction of the Skolem functions then results in a proper extension of the domain of $\mathcal{M}$. As a result, instead of obtaining a model $\mathcal{M}'$ for the Skolem form by a proper extension the interpretation valuation functions of $\mathcal{M}$, a new model has to be constructed. The upshot is that this new model $\mathcal{M}'$ is a Herbrand model for the Skolem form F^s, in this sense the result below is stronger than the corresponding usual result for FOL. However, the construction of $\mathcal{M}'$ is a bit more involved in the ambivalent case. The truth set T' of $\mathcal{M}'$ is obtained by projecting the expressions of the extended language on the expressions of the original language, and by the taking T' as the pre-image of T under this projection. The projection (translation) is the identity function on the original language.

The proof we give here is given for the case of AL, but can be generalised to FOL and intermediate cases.

1.4.2. THEOREM. (s-equivalence) *For every closed expression F in normal form, F is satisfiable iff its Skolem form is satisfiable.*

Proof: Let $F = \forall x_1 \exists y_1 \ldots \forall x_n \exists y_n.\phi$, where all outside quantifiers are indicated. Let $L_{\mathcal{G}}$ be the underlying language of F.
Let $f_1, \ldots, f_n$ be new parameters not occurring in ϕ.
Let $L' = L_{\mathcal{G} \cup \{f_1,\ldots,f_n\}}$.
Let $F^s = \forall x_1 \ldots \forall x_n.\phi\{y_1/f_1(x_1), \ldots, y_n/f_n(x_1, \ldots, x_n)\}$.
Let $\phi' = \phi\{y_1/f_1(x_1), \ldots, y_n/f_n(x_1, \ldots, x_n)\}$, that is, $F^s = \forall x_1 \ldots \forall x_n.\phi'$.

The proof of the left-to-right direction is as usual. For the converse direction, let, (by Corollary 1.4.2), $\mathcal{M} = \langle D, T \rangle$ be a Herbrand model for F; here, D consists of the closed expressions of $L_{\mathcal{G}}$. We can now, as usual, by the axiom of choice, define (external) functions $v_1, \ldots, v_n$ on D, such that for all $d_1, \ldots, d_n \in D$

$$\mathcal{M}, \{x_1/d_1, y_1/v_1(d_1), \ldots, x_n/d_n, y_n/v_n(d_1, \ldots, d_n)\} \models \phi. \qquad \text{(I)}$$

We construct, using $\mathcal{M}$ and the functions $v_1, \ldots, v_n$, a Herbrand model $\mathcal{M}' = \langle D', T' \rangle$, where D' consists of all the closed expressions in L', such that $\mathcal{M}' \models F^s$. Observe that D is a strict subset of D'.

We will define the truth-set T' using the following translation (or projection) $(\cdot)^*$ of expressions of L' into expressions of $L_{\mathcal{G}}$.

- For all $c \in \mathcal{G}$, $c^* = c$
- For all variables x, let $x^* = x$
- For f_i, choose a $c_i \in \mathcal{G}$, and let $f_i^* = c_i$
- If $t = f_i$ and $n = i$, let $(t)(t_1, \ldots, t_n)^* = v_i(t_1^*, \ldots, t_i^*)$;
- Otherwise, $(t)(t_1, \ldots, t_n)^* = (t^*)(t_1^*, \ldots, t_i^*)$.
- $(A \vee B)^* = A^* \vee B^*$ and similarly for other binary connectives
- $(\neg A)^* = \neg A^*$
- $(\forall x(A))^* = \forall x(A^*)$
- $(\exists x(A))^* = \exists x(A^*)$.

Now for all expressions t in $L_{\mathcal{G}}$, $t^* = t$. In particular, $(\phi)^* = \phi$. Also, let t be an expression in $L_{\mathcal{G}}$, and let $\sigma = \{z_1/s_1, \ldots, z_k/s_k\}$ be a substitution such that the s_i are L'-expressions. We leave it to the reader to check that $(t\sigma)^* = t^*\{z_1/s_1^*, \ldots, z_k/s_k^*\} = t\{z_1/s_1^*, \ldots, z_k/s_k^*\}$. Therefore, the following equation (II) holds:

$$\begin{aligned}
(\phi'\{x_1/d_1), \ldots, x_n/d_n\})^* &= \\
&= (\phi\{x_1/d_1, y_1/f_1(d_1), \ldots, x_n/d_n, y_n/f_n(d_1, \ldots, d_n)\})^* \\
&= \phi^*\{x_1/d_1^*, y_1/(f_1(d_1))^*, \ldots, x_n/d_n^*, y_n/(f_n(d_1, \ldots, d_n))^*\} \qquad \text{(II)} \\
&= \phi\{x_1/d_1^*, y_1/v_1(d_1^*), \ldots, x_n/d_n^*, y_n/v_n(d_1^*, \ldots, d_n^*)\}.
\end{aligned}$$

Now define the truth set T' as follows:

$$T' = \{t \in L' : t^* \in T\}$$

By induction it follows that for all sentences A in $L_{\mathcal{G}}$,

$$\mathcal{M}' \models A \quad \text{iff} \quad \mathcal{M} \models A^*.$$

In particular, by (I) and (II), $\mathcal{M}' \models F^s$. □

Another folklore result is Herbrand's theorem:

1.4.3. THEOREM. *A closed formula in Skolem form with matrix F is unsatisfiable iff there is a finite subset of the Skolem expansion of F which is unsatisfiable.*

The usual proof of the above theorem goes through, without modification, for Ambivalent Logic.

All of the above results (1.4.1 to 1.4.3) also hold for extensions of AL with any of the above-mentioned equality theories.

We have proven some of the essential ingredients to prove soundness and completeness of resolution for Ambivalent Logic: the construction of Skolem forms, the s-equivalence theorem and the Herbrand theorem. The missing ingredients for the completeness of resolution are the lifting lemma (which allows a transformation of resolution refutation on clauses in propositional logic to a resolution refutation on clauses in predicate logic), and decidability of unification for Ambivalent Logic. The traditional proof of the lifting lemma goes through, without modification, for Ambivalent Logic. Unification theory for Ambivalent Logic will be discussed in Section 1.5. The traditional proof of soundness of resolution for FOL (see for instance Schöning [Sch89]) goes through, without modification, for Ambivalent Logic. We conclude that, modulo the results on unification in the next section, we have the following result:

1.4.4. THEOREM. *Resolution is a sound and complete inference method for AL.*

We conclude this section by proving one more result. Given the fact that AL is a syntactic extension of FOL, and has the same derivational calculus, the question of conservativity of AL over FOL arises, that is: if a formula ϕ in standard syntax is derivable in AL, is it then also derivable in FOL — and vice versa? The answer, as the following theorem shows, is affirmative. This result shows that AL is an extension of FOL. The proof uses both directions of the above s-equivalence theorem 1.4.2. The technique used in the proof is similar to that used in the proof of Theorem 1.4.2: from a model, a new, ambivalent model is defined, the truth-set of which is a pre-image of a syntactic projection to the truth-set of the original model. In this case, a Herbrand model for a language with standard syntax, is extended to an elementary equivalent Herbrand model for the associated ambivalent language.

1.4.5. THEOREM. *Let ϕ be a formula in standard syntax. Then $\models_{FOL} \phi$ iff $\models_{AL} \phi$.*

Proof: The left-to-right direction follows from the fact that every derivation in FOL is also a derivation in AL, combined with completeness of FOL, and soundness of AL. For the converse direction (conservativity of AL over FOL), it suffices, by contraposition, to show that if ϕ is satisfiable in FOL, then it is also satisfiable as an AL formula.

So let ϕ be satisfiable in FOL. Without loss of generality we can assume that ϕ is in normal form. By the s-equivalence theorem for FOL, the Skolem form ϕ^s is also satisfiable. In particular, there is a Herbrand model $\mathcal{M} = \langle D, T\rangle$ satisfying ϕ^s.

Using $\mathcal{M}$, we will construct an ambivalent Herbrand model $\mathcal{M}' = \langle D', T' \rangle$ satisfying ϕ^s.

Let C, F, and P be the set of constants, respectively functions, respectively relation symbols occurring in ϕ^s. Then the universe D of $\mathcal{M}$ is generated by $\langle C, F \rangle$. Let D' be the ambivalent universe generated by the parameter set $C \cup F \cup P$. In order to define the truth set T', we will make use of a (total and surjective) function $(\cdot)^*$ from D' to D. (The definition of * is inspired by the abstraction function defined in Chapter 2; here, we use, instead of fresh constants, some element of D.)

Choose an arbitrary $d \in D$. Define $^* : D' \to D$ as follows:

1. For $c \in C \cup F \cup P$,
   $c^* := c$ if $c \in C$
   $c^* := d$ otherwise.
2. for closed terms $(t)(t_1, \ldots, t_n)$,
   $((t)(t_1, \ldots, t_n))^* := t(t_1{}^*, \ldots, t_n{}^*)$ if t is an n-ary predicate in P
   $(t)(t_1, \ldots, t_n)^* := d$ otherwise.
3. For all other $t \in D'$, $t^* := d$.

Observe that, for every $t \in D$, $t^* = t$.

Using the function *, the truth set T' of $\mathcal{M}'$ can now be defined as follows:

$$T' = \{t \in D' : t^* \in T\}.$$

Now let $\phi^s = \forall x_1 \ldots \forall x_n.A$, where A is a conjunctive normal form.

$\mathcal{M} \models_{FOL} \forall x_1 \ldots \forall x_n.A$
$\Longrightarrow$ {by definition of the satisfaction relation}
for all $d_1, \ldots, d_n \in D$ $\mathcal{M} \models_{FOL} A\{x_1/d_1, \ldots, x_n/d_n\}$
$\Longrightarrow$ {by definition of * and by the form of A}
for all $d_1, \ldots, d_n \in D'$, $\mathcal{M} \models_{FOL} A\{x_1/d_1{}^*, \ldots, x_n/d_n{}^*\}$
$\Longrightarrow$ {by definition of T'}
for all $d_1, \ldots, d_n \in D'$, $\mathcal{M}' \models_{AL} A\{x_1/d_1, \ldots, x_n/d_n\}$
$\Longrightarrow$ {by definition of the satisfaction relation}
$\mathcal{M}' \models_{AL} \forall x_1 \ldots \forall x_n.A$.

From the above implication and the easy side of the s-equivalence theorem 1.4.2, it immediately follows that $\mathcal{M}'$ is a closed term model satisfying ϕ. □

The soundness direction of the above theorem generalises to AL + ET(I), AL + ET(I,II), and AL + ET. The completeness side however, trivially does not hold for either of AL + ET(I) and AL + ET(I,II), as FOL incorporates all of ET.

Although AL + EQ(I,II) is not conservative over FOL for formulas with equality, this logic is of some interest in the context of intensional logic. By the soundness and completeness w.r.t. the appropriate semantics, it does not satisfy ET(III) — therefore, it is opaque w.r.t. substitutions of equal terms. In addition, by the ambivalent syntax of AL, which allows for all expressions to occur in term positions, AL + EQ(I,II) has identity naming, that is, expressions can be represented by themselves.

In the domain of knowledge and belief this is a useful property. In contrast, AL + ET is, by the above theorem, a conservative extension of FOL.

A language with ambivalent syntax was first defined by Richards [Ric74] in the context of intensional predicate logics. Kowalski and Kim [KK91] advanced ambivalent language as appropriate for the Vanilla meta-interpreter, especially for extensions of it in the field of meta-logic programming for muti-agent knowledge, and consequently, Richards [Ric74] has become a standard reference in the literature on the Vanilla meta-interpreter. It should be observed however, that Richards' syntax was devised for a different purpose and has a calculus adapted to this goal. As a result, standard first order logic with ambivalent syntax is unsound with respect to the semantics proposed in Richards [Ric74], as the following example shows. Let $\mathcal{L}$ be an ambivalent language with one binary relation symbol R, one unary function symbol f, and one constant symbol c. Then $\forall x\, R(x, Q(x)) \rightarrow \forall x\, R(f(x), Q(f(x)))$ should be valid in every model. However, one can construct a model M according to Richards' definitions which validates the antecedent, and in which the consequent is false. Take, according to the definitions in Richards [Ric74], as the domain of M, $D_M = \{c\} \cup \{\phi : \phi \in \text{CLFORM}_{\mathcal{L}}\}$. The extension of Q can be taken arbitrary. $V(f)$ is the identity function on D_M. The extension of R is taken as follows: $V(R) = \{\langle d, Q(d)\rangle | d \in D_M\}$. By Richards' definition, the interpretation function * is the identity function on constants and closed formulas. Thus, M validates $\forall x\, R(x, Q(x))$. However, $(f(c))^* = V(f)(c^*) = c$, while $(Q(f(c)))^* = Q(f(c))$. So $\langle (f(c))^*, (Q(f(c)))^*\rangle = \langle c, Q(f(c))\rangle$, which does not belong to $V(R)$.

1.5 Unification

In the present section we show how the Martelli-Montanari unification algorithm can be adapted to the case of ambivalent syntax. We show that the adapted algorithm (which is defined on page 25) has all the desired properties, in particular, termination and, in case of successful termination, generation of unifiers which are most general within an appropriate class of unifiers. We follow the outlines of the theory for unification as given in Doets [Doe94], to which we also refer the reader for the usual definitions of unifier and most general unifier.

There are several differences between unification for standard syntax and unification for full ambivalent syntax.

1. In ambivalent syntax, functions and predicates are arityless. Unification of two atoms with different arities has to be excluded. An extra action (9) is sufficient: halt with failure on an equation $(t)(t_1, \dots, t_n) = s(s_1, \dots, s_m)$, if $n \neq m$.
2. In the Martelli-Montanari algorithm for standard syntax, there are two actions for functional atoms:
   halting with failure on $f(t_1, \dots, t_n) = g(s_1, \dots, s_n)$, if f is unequal to g; replacement with $t_1 = s_1, \dots, t_n = s_n$, otherwise.
   In ambivalent syntax, all expressions can occur in function positions. The above two actions are replaced by one single action (8), accounting for the unification of the expressions in the function positions as well as the arguments.

An extra action (10) is needed to prevent unification of functional atoms with expressions that are not functional atoms or variables. In addition, two extra actions (6) and (7) deal with unification of parameters.

3. In ambivalent syntax, conjunctive, disjunctive, implicational, and negated expressions occur as terms and as subexpressions, thus they are candidates for unification. For example, $x \vee y$ and $a \vee c(z)$ are unified by $\{x/a, y/c(z)\}$. This is accounted for in the unification algorithm by the actions (11) and (13). Appropriate failure conditions are reflected by the actions (12) and (14).
4. Likewise, quantified expressions have to be dealt with. For closed quantified expressions, semantical identity coincides with syntactical identity. However, quantified expressions in general are still liable for unification. First, they can unify with variables. For example, $\exists x(x)$ and y are unified by $\{y/\exists x(x)\}$. Further, quantified expressions can contain free variables, which are candidates for unification. As an example, $\exists x(x(y))$ and $\exists x(x(c))$ are unified by $\{y/c\}$. In contrast, $\exists x(x)$ and $\exists y(y)$ cannot be unified, as both are closed expressions.
   Unification of quantified expressions is partly engineered by action (15), which eliminates identical quantifiers: $\exists x.t = \exists x.s$ is repaced by $t = s$, and similarly for the universal quantifier. In contrast, the semantic distinction between $\exists x.t$ and $\exists y.t\{x/y\}$ is reflected in action (16), which halts with failure on equations $\exists x.t = \exists y.s$, if x and y are different (and likewise for the universal quantifier). However, the (necessary) action (15) might lead to incorrect results, as it releases bound variables, which subsequently become, incorrectly, candidates for non-trivial unification. As an example, action (15) replaces the unsolvable equation $\exists x.x = \exists x.c$ with the solved equation $x = c$. (Below we will formally define the notion of solved equation.) This problem is solved in two ways.
   First, before the algorithm is run on a set of equations, all the bound variables should be renamed to marked variants. An appropriate renaming function will be given in Definition 1.5.1 below. This enables us to keep track of the origin of variables during execution of the algorithm.
   Second, the algorithm should treat marked and unmarked variables differently. In particular, marked variables, which should be thought of as bound variables, only unify with themselves and with unmarked variables. Trivial unification of marked and unmarked variables is dealt with in action (1). Non-trivial unification of marked variables with anything but unmarked variables is excluded by action (2). However, unification of unmarked variables with expressions in which marked variables occur free, should be possible. For example, consider the expressions $\exists x(p(x) \vee q(x))$ and $\exists x(y \vee q(x))$, where y is an unmarked variable. These expressions are unified by $\{y/p(x)\}$. That is, unification of unmarked variables with expressions in which marked variables occur free, should be allowed. This is taken care of by action (4). The usual actions (3) and (4) only apply if the left-hand variables are unmarked.

The appropriate marking of bound variables is obtained by applying a marking function $(\cdot)^m$, which replaces all occurrences of bound variables with marked copies of these variables. The effect of the marking function is a renaming of the bound

Martelli-Montanari unification algorithm for ambivalent syntax

```
1   $x = x$                    where $x$ is a (marked or unmarked) variable
      remove
2   $x' = t$                   where $x'$ is a marked variable,
                               $t$ is not an unmarked variable,
                               and $t$ is different from $x'$
      halt with failure
3   $x = t$                    where $x$ is unmarked, $t$ is different from $x$,
                               and $x$ occurs in $t$
      halt with failure
4   $x = t$                    where $x$ is unmarked, $t$ is different from $x$,
                               and $x$ does not occur in $t$
      replace $x$ by $t$ in all other equations
5   $t = x$                    where $t$ is not an unmarked variable
      replace by $x = t$
6   $c = c$                    where $c$ is a parameter
      remove
7   $c = t$                    where $c$ is a parameter, $t$ is not a variable
                               and $c$ and $t$ are different
      halt with failure
8   $(t)(t_1, \ldots, t_n) = (s)(s_1, \ldots, s_n)$
      replace by $t = s, t_1 = s_1, \ldots, t_n = s_n$
9   $(t)(t_1, \ldots, t_n) = (s)(s_1, \ldots, s_m)$     where $n \neq m$
      halt with failure
10  $(t)(t_1, \ldots, t_n) = s$   where $s$ is not a variable or a functional atom
      halt with failure
11  $\neg t = \neg s$
      replace by $t = s$
12  $\neg t = s$               where $s$ is not a variable or a negated expression
      halt with failure
13  $t_1 \diamond t_2 = s_1 \diamond s_2$      where $\diamond$ is one of the binary connectives
      replace by $t_1 = s_1, t_2 = s_2$
14  $t_1 \diamond t_2 = s$            where $s$ is not a variable or of the form $s_1 \diamond s_2$
      halt with failure
15  $Qx'(t) = Qx'(s)$          where $Q$ is one of the quantifiers $\exists$ and $\forall$
      replace by $t = s$
16  $Qx'(t) = s$               where $Q$ is one of the quantifiers $\exists$ and $\forall$,
                               $s$ is not a variable and $s$ is not of the form $Qx'(v)$
      halt with failure
```

variables in an expression, with the effect that no variable occurs both bound and free after marking.

Before we define the marking function, we introduce some notation. Recall that $(t)(t_1, \ldots, t_n)$ is a functional atom with n-ary predicate (or function) t, and arguments t_i. We will use the notation $t(x_1, \ldots, x_n)$ to indicate an expression t for which $FV(t) \subseteq \{x_1, \ldots, x_n\}$.

1.5.1. DEFINITION. (Marking bound variables)

$$\begin{array}{ll} c^m = c & \textit{for parameters } c, \\ x^m = x & \textit{for unmarked variables } x, \\ x'^m = x' & \textit{for marked variables } x', \\ ((t)(t_1, \ldots, t_n))^m = (t^m)(t_1^m, \ldots, t_n^m) & \end{array}$$

$$\begin{array}{ll} (\exists x(t))^m = \exists x'(t\{x/x'\})^m & \textit{where } x \textit{ is an unmarked variable}, \\ (\exists x'(t))^m = \exists x'(t^m) & \textit{where } x' \textit{ is a marked variable}, \end{array}$$

and similarly for universally quantified expressions.
In addition, $(\cdot)^m$ *commutes with the logical connectives.* □

The unification algorithm for AL will only yield the desired results if applied to terms in which all the bound variables are marked.

We will use the following terminology:

1.5.2. DEFINITION. An expression t is *clean* if $t^m = t$. A set of equations $\{t_1 = s_1, \ldots, t_n = s_n\}$ is clean if all of the t_i and s_i are clean. A pair of sequents $\langle t_1, \ldots, t_n \rangle$, $\langle s_1, \ldots, s_n \rangle$ is clean if the associated set of equations $\{t_1 = s_1, \ldots, t_n = s_n\}$ is clean. □

In particular, marked variables can occur free in clean expressions, but in contrast, the latter do not contain bound occurrences of unmarked variables.

In the correctness proof of the algorithm we need the following notions of ambivalent substitution and ambivalent unifier:

1.5.3. DEFINITION. σ is an *ambivalent substitution* if no marked variables occur in the domain of σ. □

1.5.4. DEFINITION. Let t and s be expressions. σ is an *ambivalent unifier* for t and s, if σ is an ambivalent substitution and $t\sigma = s\sigma$. □

Observe that not every unifier is also an ambivalent unifier. For example, while $\{x'/f(y)\}$ unifies $g(x')$ with $g(f(y))$, no ambivalent unifiers exist for this tuple. In contrast, every ambivalent unifier is a unifier.

1.5.5. PROPOSITION. *Let* $L = \langle t_1, \ldots, t_n \rangle$ *and* $R = \langle s_1, \ldots, s_n \rangle$ *be two sequents of clean expressions such that no marked variables occur free in any of the* t_i *and* s_i. *Then* L *and* R *are unifiable iff there is an ambivalent unifier for* L *and* R.

Proof: Suppose σ unifies L and R. Now let τ be the restriction of σ to $FV(L) \cup FV(R)$. Then, by assumption, τ is an ambivalent unifier of L and R. For the other direction, notice that every ambivalent unifier is a unifier. □

We relativise the notion of most general unifier to the class of ambivalent unifiers.

1.5.6. DEFINITION. An ambivalent unifier θ for a set of (clean) expressions E is a *most general ambivalent unifier* for E (or, in short, an m.g.a.u.) if for every ambivalent unifier σ for E there is an ambivalent substitution τ such that $\sigma = \theta\tau$. An m.g.a.u. θ for E is *strong* if for every ambivalent unifier σ of E, $\sigma = \theta\sigma$. □

An m.g.a.u. is not necessarily an m.g.u., as the following counterexample witnesses.

1.5.1. COUNTEREXAMPLE. Let $\theta = \{y/f(x')\}$. It is easy to check that θ is an m.g.a.u. for $E = \{p(\exists x'.f(x')) = p(\exists x'.y)\}$. Also, $\sigma = \{y/f(x'), x'/y\}$ is a unifier of E. However, suppose there is a substitution τ such that $\sigma = \theta\tau$. Then $f(x')\tau = f(x')$, and thus $x' \notin dom(\tau)$. But this contradicts $\{x'/y\} \in \{y/f(x')\}\tau$.

In addition, we need to adapt the notions of solved equation and equivalence between sets of equations.

1.5.7. DEFINITION. A set of equations $\{t_1 = s_1, \ldots, t_n = s_n\}$ is *solved* if
1. the t_i are pairwise different, unmarked variables, and
2. no t_i occurs in any of the s_j. □

A solved set of equations $E = \{x_1 = s_1, \ldots, x_n = s_n\}$ determines a most general ambivalent unifier $\{x_1/s_1, \ldots, x_n/s_n\}$ for E (or, more precisely, for the associated pair of sequences $\langle x_1, \ldots, x_n\rangle$ and $\langle s_1, \ldots, s_n\rangle$). We need one more definition.

1.5.8. DEFINITION. Two sets of equations of ambivalent terms are *equivalent* if they have the same ambivalent unifiers. □

Now we are in position to prove correctness of the unification algorithm.

1.5.2. THEOREM. *The Martelli-Montanari unification algorithm for ambivalent syntax, when applied to a finite set of clean equations, results in a solved set of equations, determining a strong most general ambivalent unifier for the associated sequences in case an ambivalent unifier exists, and terminates with failure otherwise.*

Proof: The theorem follows from the following claims.

1. CLAIM. *Every non-halting action applied to a set of clean equations produces an equivalent set of clean equations.*

Proof: None of the actions 1, 4, 5, 6, 8, 11, 13, and 15 introduces a bound, unmarked variable. Therefore any of these actions transforms a set of clean equations into a new set of clean equations.

Preservation of equivalence is trivial for the actions 1, 5, 6, 8, 11, and 13. For action 4, preservation of equivalence is proven as usual.
For action 15, let σ be an ambivalent substitution. Then the following holds:

$$\begin{aligned}
&(\exists x'(t))\sigma = (\exists x'(s))\sigma \iff \qquad \{x' \notin dom(\sigma)\}\\
&\exists x'(t\sigma) = \exists x'(s\sigma) \iff t\sigma = s\sigma.
\end{aligned}$$ □

2. CLAIM. *The algorithm terminates.*

Proof: Consider the lexicographic order $<_3$ on $\mathbb{N}^3$. That is,

$$(n_1, n_2, n_3) <_3 (m_1, m_2, m_3)$$

iff

$$\begin{aligned} & n_1 < m_1 \\ \text{or}\quad & n_1 = m_1 \ \& \ n_2 < m_2 \\ \text{or}\quad & n_1 = m_1 \ \& \ n_2 = m_2 \ \& \ n_3 < m_3. \end{aligned}$$

Given a set of equations E, we call an unmarked variable x *solved in* E if, for some expression t, $x = t \in E$, and this is the only free occurrence of x in E. We call a variable x *unsolved in* E if x is unmarked and x is not solved in E.

With each set of clean equations E we now associate the following three functions:

$uns(E)$:= the number of unsolved variables in E,
$lfun(E)$:= the total number of occurrences of parameter symbols on the left hand side of the equations in E,
$lsym(E)$:= the total number of symbols (including brackets) occurring on the left hand side of equations of E.

We claim that each of the non-halting actions of the algorithm strictly reduces the triple $(uns(E), lfun(E), lsym(E))$.

Indeed, action 4 decreases $uns(E)$ by 1, while none of the successful actions increase $uns(E)$. Also, none of the other successful actions increase $lfun(E)$. Action 5 decreases $lfun(E)$ by 1 if t is a parameter, and decreases $lsym(E)$ by at least 1 otherwise. The actions 1, 6, 8, 11, 13, and 15 all decrease $lsym(E)$. Termination of the algorithm now follows from the well-foundedness of $<_3$.
□

3. Claim. *If the algorithm terminates successfully on a set of clean equations, then the final set of equations is solved.*

Proof: Suppose the algorithm terminates successfully, resulting in a set of clean equations E. Then none of the actions 5 – 16 applies to E, so the left hand expressions are all variables. Action 2 does not apply to E, so these are all unmarked variables. Finally, actions 1, 3, and 4 do not apply to E, so none of these variables occurs on the right hand side of any of the final equations. □

4. Claim. *If the algorithm halts with failure on a clean set of equations, then this set does not have an ambivalent unifier.*

Proof: Suppose the algorithm halts on the clean set E. If failure is the result of action 2, then the equation $x' = t$, where t is different from x' and t is not a variable, is a member of E. Clearly there are no ambivalent unifiers for $x' = t$. If failure is the result of action 3, then $x = t$ is an element of E, and there is no unifier σ for x and t, as $x\sigma$ is a proper subterm of $t\sigma$. If failure is the result of action 7, then $c = t$ is an element of E, and there is no unifier σ for c and t, because $c\sigma = c$, and $t\sigma$ is either a parameter different from c or an expression that is neither a variable nor a parameter. In the other cases, similar arguments apply. □
This completes the proof of Theorem 1.5.2. □

Clearly, this result is less general than the corresponding result for the original version of the unification algorithm. First of all, it applies only to those equations in which the sets of free and bound variables are distinct. It is an open question whether this restriction can be dropped. Second, the unifiers generated are most general only within the class of ambivalent unifiers, that is, those unifiers for which the domain does not contain any variable that occurs bound in the original set of equations. Counterexample 1.5.1 above suggests that this restriction can not be dropped.

As a corollary of the above theorem we now have a decidable unification algorithm for Prolog syntax. Clearly, in the Prolog case, where quantified expressions do not occur in term positions, we do not have to deal with the renaming of bound variables. The relevant actions in the Prolog version of the unification algorithm are the *Prolog actions* 1, 3, 4, 5, 6, 7, 8, 9, and 10. We leave it to the reader to check that the following holds.

1.5.3. THEOREM. *The Martelli Montanari unification algorithm for Prolog's syntax, consisting of the Prolog actions 1, 3, 4, 5, 6, 7, 8, 9, and 10 results in a solved set of equations, determining a strong most general unifier in case a unifier exists, and terminates with failure otherwise.*

1.6 Comparison with Hilog

Hilog (Chen et al. [CKW93]) was developed as a language for higher order Logic Programming. In the discussion below, we assume that the reader is familiar with Hilog. We will here mainly point out some of the differences and similarities between Hilog and AL.

AL syntax is an extension of the syntax of Hilog. While the syntax of AL is fully ambivalent, Hilog syntax is characterised by the occurrence of terms in formula-, function- and predicate positions. (For example, $(c(a))(c)$, $x \rightarrow p(x)$, and $\forall x \exists y.(x(p))(y)$ are Hilog formulas.) Other than AL, Hilog does not admit quantified expressions in any unusual positions. As an example, $p(\forall x p(x))$ is an AL expression, but not a well-formed Hilog expression. In either syntax, the parameters are arityless.

While syntactically AL can be considered an extension of Hilog, the two logics differ considerably in semantics. The distinguishing feature of the (first-order) Hilog semantics is that each parameter of the language has a unique intension—that is, the interpretation function associates with each individual parameter (regardless of its contextual role), exactly one object in the domain of a model. With each intension then, several extensions are associated that capture the different contextual roles of the parameter. This extends to general terms. As a consequence, the schema $\forall x \forall y.(x = y \rightarrow \phi(x) \leftrightarrow \phi(y))$ is true in Hilog's semantics, and also $\forall x \forall y \forall z.(x = y \rightarrow x(z) \leftrightarrow y(z))$. In contrast, in the context of AL we have a *choice* between validating the above schema or not, by either taking all of ET as the equality theory, or restricting the equality to ET(I,II). Thus, unlike AL, Hilog is not appropriate for intensional logics, where opaqueness is usually desirable. (In contrast, observe that equivalence of terms does not imply their equality in neither Hilog nor AL.)

Another difference between AL and Hilog is found in comparing their respective relations to FOL. As we have seen, AL is a conservative extension of FOL without equality, and AL with the usual equality theory ET is a conservative extension of FOL with equality (Theorem 1.4.5). In Hilog, conservativity over FOL is restricted to the classes of equality free formulas and definite clauses in which equality only occurs in body-atoms. A standard derivational calculus for FOL is sound and complete w.r.t. AL semantics (Theorem 1.4.1). In contrast, in accordance to the intension of developing Hilog as basis for Logic Programming, paramodulation is introduced in [CKW93] as a derivational calculus for Hilog, and its soundness and completeness with respect to the Hilog semantics is proven. It is also shown that Hilog can be encoded in FOL, and a standard derivational calculus for FOL can be soundly used for the derivation of encoded Hilog formulas.

It should be noted that, despite the second order aspects of Hilog and AL syntax, both are essentially first order theories. In higher order predicate logic, the second order variables range over all relations over the intended domain. That is, in higher order semantics, functions and predicates are identified with their domain. In contrast, in Hilog and AL, the 'second order' variables range over elements of the intended domain. As a result, relation comprehension does not generally hold in AL and Hilog. That is, $\exists p \forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow \phi(x_1, \ldots, x_n))$ is not generally valid in Hilog and AL. For example, relation comprehension is not true in AL for $\exists z q(x, y, z)$. But, unlike in Hilog, both $\exists p \forall x (p(x) \leftrightarrow \neg q(x))$ and $\exists p \forall x (p(x) \leftrightarrow \exists u (q((u))(x))$ are valid in AL.

A consequence of the identification of functions and predicates with their extensions in higher-order logic, is that the undecidability of this extensional equality carries over to the unification problem. Both for AL and Hilog however, unification is decidable.

1.7 Conclusion

The results reported show that, with minor modifications, basic proof theoretic results for first order predicate logic also go through for Ambivalent Logic. In particular, unification for AL is decidable, and both a standard derivational calculus and resolution are sound and complete inference methods for AL. A conservativity result shows that AL should be considered as a (syntactic) extension of first order predicate logic. All of the results reported relativise to subsystems of Ambivalent Logic that are frequently used in practice, such as Prolog syntax and the syntax(es) used in Vanilla meta-programming and data bases. In particular, our results justify the current practice of using ambivalent syntax in Prolog and Vanilla meta-programming. In addition, various properties of AL, such as the optional opaqueness with respect to equality and the flexibility of its syntax, suggest that AL may provide an interesting format for the representation of knowledge and belief.

Note

This chapter will appear as [KJ95] "A Vademecum of Ambivalent Logic", M. Kalsbeek and Y. Jiang, in: Meta-Logics and Logic Programming, Eds. K.R. Apt and F. Turini, The MIT Press, 1995, pp. 27–56.

Part II

Meta-Logic Programming

Chapter 2

Correctness of the Vanilla meta-interpreter

We discuss and compare various approaches to correctness of the Vanilla meta-interpreter (procedural, declarative, S-semantics). We compare the typed meta-interpreter with the untyped version, and argue that the procedural correctness proof for the typed interpreter has great generality. We present a detailed proof of declarative correctness in the context of the ambivalent syntax, which is the appropriate syntax underlying most amalgamated extensions of the Vanilla meta-interpreter.

2.1 Introduction

In this chapter we study the simplest meta-interpreter for definite logic programs, usually known as the Vanilla meta-interpreter. In particular, we focus on correctness results for the Vanilla meta-interpreter. The Vanilla meta-interpreter is a definite logic program which consists of two parts: a general part **V**, which consists of an intensional formalisation of derivability by SLD-resolution from definite object programs, and an object program specific part meta-P, which consists of a meta-level description of the clauses of an object program P.

2.1.1. DEFINITION. The standard *Vanilla meta-interpreter* $\mathbf{V}_P$ for definite object programs P.

V	[M1]	$demo(empty) \leftarrow$
	[M2]	$demo(x\&y) \leftarrow demo(x), demo(y)$
	[M3]	$demo(x) \leftarrow clause(x,y), demo(y)$
meta−P	[M4]	$clause(A, B_1\& \dots \&B_n) \leftarrow$ for every clause $A \leftarrow B_1, \dots, B_n$ in P
	[M5]	$clause(A, empty) \leftarrow$ for every clause $A \leftarrow$ in P

□

In the above, $B_1\& \dots \&B_n$ is an abbreviation for

$B_1\&(B_2\& \cdots (B_{n-1}\&B_n)\cdots)$, if $n > 1$,
B_1 if $n = 1$.

Additionally, a meta-interpreter for normal programs is obtained by extending $\mathbf{V}_P$ with the clause

$$[\text{M}\neg] \quad demo(not\ x) \leftarrow \neg\, demo(x).$$

In the context of the present chapter however, we mainly concentrate on the interpreter for definite object programs.

A correctness result for the Vanilla-meta interpreter establishes that its intended behaviour is its observed behaviour. Basically, a correctness result for $\mathbf{V}_P$ is a relation between $\mathbf{V}_P$ and P of the form

$$\mathbf{V}_P \mid\!\sim demo(A) \qquad \text{iff} \qquad P \mid\!\sim A,$$

where $\mid\!\sim$ is a semantical consequence relation w.r.t. a preferred semantics (declarative correctness), or a relevant derivational consequence relation such as refutability by SLD(NF)-resolution (procedural correctness). The two directions of the abstract correctness relation are usually distinguished as soundness (from left to right) and completeness (vice versa). Various correctness results and their proofs will be discussed and compared in Section 2.2. In addition, we will present one specific correctness proof in some detail in Section 2.4.

One of the problems involved in proving correctness results for the Vanilla meta-interpreter originates in the fact that this program is not well-typed. As an example, consider the following object program Q:

$$\begin{array}{l} p(c) \leftarrow \\ r(x) \leftarrow \\ q(x) \leftarrow p(x) \end{array}$$

The associated Vanilla meta-interpreter $\mathbf{V}_Q$ is

$$\begin{array}{l} demo(empty) \leftarrow \\ demo(x\&y) \leftarrow demo(x), demo(y) \\ demo(x) \leftarrow clause(x,y), demo(y) \\ clause(p(c), empty) \leftarrow \\ clause(r(x), empty) \leftarrow \\ clause(q(x), p(x)) \leftarrow \end{array}$$

Now consider the variables which occur in $\mathbf{V}_Q$ above. The variables occurring in the second and third clause are intended to range over (conjunctions of) object level atoms, whereas the variables that occur in the last three clauses are meant to range over the domain of the object program. Thus, intuitively, the Vanilla meta-interpreter is a typed program with two types: the variables occurring in the clauses [M2] and [M3] are intended as meta-variables ranging over object level atoms; the variables that occur in the part which represents the object level program, are meant to be object level variables ranging over object level terms.

Motivated by the observation that the intuitive interpretation of the Vanilla meta-interpreter is typed, Hill and Lloyd [HL88] advocate a typed version of the Vanilla meta-interpreter and prove its (declarative and procedural) correctness. We will discuss this correctness result for the typed version of $\mathbf{V}_P$ in some detail in Section 2.2. However, the *untyped* version of the Vanilla meta-interpreter, and extensions of it, are frequently used in general Prolog practice and in applications (see, for instance, programs discussed in Sterling and Shapiro [SS86] and Kowalski and Kim [KK91]). Therefore, a correctness result for the untyped interpreter is of interest.

Typically, the untyped Vanilla meta-interpreter does not distinguish between object- and meta-level variables and terms. As a result, the least Herbrand model of $\mathbf{V}_Q$ above contains 'unrelated' atoms like $demo(r(empty))$ and $demo(r(r(c)\&p(c)))$, while the least Herbrand model of the object program Q does not contain the atoms $r(empty)$ and $r(r(c)\&p(c))$. (Unrelated atoms are meta-level atoms of the form $demo(p(t))$, where p is a predicate from the object program while t is a term which does not belong to the language of the object program — therefore, they do not correspond to an atom of the object level). This example illustrates that a declarative correctness result for the Vanilla meta-interpreter will, in general, not establish a complete correlation between the least Herbrand models of meta- and object programs. We will investigate in full detail, in Section 2.4, the exact correspondence between these least Herbrand models. In particular, we will show that the unrelated atoms occuring in the least Herbrand model of Vanilla can be given a useful interpretation.

Another issue is the representation of object level predicates and function symbols in the meta-program. In principle, two options are available. First, the function symbols f and relation symbols p that occur in the object program can be represented by function symbols f' and p' in the associated meta-program. Using this *functional representation*, object level clauses $A \leftarrow B_1, \ldots, B_n$ are represented in the meta-program as $clause(A', B_1', \ldots, B_n') \leftarrow$. For a more detailed discussion of the functional representation and a precise definition of the Vanilla meta-interpreter in this context we refer to Martens and De Schreye [MS95a].

Second, the naive *identity representation* can be used. In that case, the clauses of the object program are represented as in Definition 2.1.1 above. As long as the Vanilla meta-interpreter is not combined with (clauses from) the object program or extended with *amalgamated* clauses (in which atoms from both the object-level and the meta-level occur), the identity representation we use in Definition 2.1.1 is just a special case of the functional representation — every symbol from the alphabet of the object language being represented by itself. However, in the case of amalgamated extensions, the underlying language of the meta-program is non-standard. As an example, consider an extension of $\mathbf{V}_Q$ above with the object program Q and the clause $demo(q(f(x)) \leftarrow q(x), demo(p(x))$. The non-standard syntax employed here is characterised by the occurrence of atoms in term positions, obtained by allowing predicates to occur in function positions (overloading). This is the only form of ambivalence we will encounter in the context of the present chapter. The Vanilla meta-interpreter of Definition 2.1.1, which uses the identity representation of the object level language, is typically suitable for applications in which it is extended with object clauses and amalgamated clauses such as the above. (Again, we refer to Sterling and Shapiro [SS86] and Kowalski and Kim [KK91] for examples of applications employing amalgamated extensions or modifications of Vanilla.) In contrast, the version using a non-identity functional representation is obviously not suitable for such extensions.

Given the interest of amalgamated extensions of the Vanilla meta-interpreter, and in view of the fact that Prolog itself employs ambivalent, rather than standard syntax, an account of meta-logic programming in the context of ambivalent syntax is of interest. This chapter provides an exploration in this area.

Outline

Section 2.2 provides an overview of the various existing correctness results for the Vanilla meta-interpreter. In addition, this section contains a proof sketch of the procedural correctness of the untyped Vanilla meta-interpreter for normal object programs.

In Section 2.4 we prove a satisfying declarative correctness result for the untyped Vanilla meta-interpreter for definite object programs in the context of ambivalent syntax. Section 2.3 provides some technical preliminaries for this correctness proof.

In Section 2.5 we discuss a simple amalgamation.

Conventions

We will adhere to the following conventions regarding syntax and language of programs.

The underlying language of an object program P is identified with $\mathcal{L}_P$, the language generated by $(\mathcal{R}_P, \mathcal{F}_P, \mathcal{C}_P)$ and standard syntax, where $\mathcal{R}_P$, $\mathcal{F}_P$, and $\mathcal{C}_P$ are, respectively, the set of relation symbols, function symbols and constants occurring in P.

For the underlying language $\mathcal{L}_{V_P}$ of the associated meta-program $\mathbf{V}_P$ we take the language generated by $(\mathcal{R}_{V_P}, \mathcal{F}_{V_P}, \mathcal{C}_{V_P})$, where $\mathcal{R}_{V_P} = \mathcal{R}_P \cup \{demo(\cdot), clause(\cdot,\cdot)\}$, $\mathcal{F}_{V_P} = \mathcal{F}_P \cup \{(\cdot)\&(\cdot)\}$, and $\mathcal{C}_{V_P} = \mathcal{C}_P \cup \{empty\}$. In the case of the Vanilla meta-interpreter for normal programs, the set of function symbols $\mathcal{F}_{V_P}$ is extended with $neg(\cdot)$. Unless otherwise specified, we assume that the syntax of $\mathbf{V}_P$ is ambivalent.

In the present context, we do not need a fully ambivalent syntax. A level of ambivalence allowing atoms to occur in term positions is appropriate for (amalgamated extensions of) the Vanilla meta-interpreter. More precisely, we will be concerned with ambivalent term-languages $\mathcal{L}_{amb}$, for which the set of atoms $\text{ATOM}_{\mathcal{L}_{amb}}$ and the set of terms $\text{TERM}_{\mathcal{L}_{amb}}$ are given by the following definition. We assume that the language is generated by predicates R_L, function symbols F_L, and constants C_L.

2.1.2. DEFINITION.

$\text{ATOM}_{\mathcal{L}_{amb}}$ (1) $p(t) \in \text{ATOM}_{\mathcal{L}_{amb}}$ is an atom if $p \in R_L$ and $t \in \text{TERM}_{\mathcal{L}_{amb}}$;
(2) there are no other atoms.

$\text{TERM}_{\mathcal{L}_{amb}}$ (1) $c \in \text{TERM}_{\mathcal{L}_{amb}}$ if $c \in C_L$;
(2) $x \in \text{TERM}_{\mathcal{L}_{amb}}$ if x is a variable;
(3) $t \in \text{TERM}_{\mathcal{L}_{amb}}$ if $t \in \text{ATOM}_{\mathcal{L}_{amb}}$;
(4) $f(t) \in \text{TERM}_{\mathcal{L}_{amb}}$ if $t \in \text{TERM}_{\mathcal{L}_{amb}}$ and $f \in F_L$;
(5) there are no other terms. □

A justification of this variant on the usual syntax for first order predicate logic is given in the previous chapter on Ambivalent Logic.

Other conventions regarding the language underlying the meta-program are possible. For example, the logical connective & , which we choose to represent as a function symbol in the language of the meta-programs, could also be represented as a relation symbol. Also, more general levels of ambivalence can be allowed in the syntax. It should be stressed that none of these particular choices affects the results reported below.

We use some of the basic concepts and results of the theory of logic programs. We adhere to the terminology used in Lloyd [Llo87]. In particular, $\mathbf{M}_P$ will indicate the least Herbrand model of a program P, and $\mathbf{B}_P$ will indicate the Herbrand base. The set of free variables of a term t will be indicated by FV(()t). As observed in Martens and De Schreye [MS95a] and witnessed by the results of the previous chapter, the basic theory for definite programs as developed in Lloyd [Llo87] holds without any restriction for programs with ambivalent language, and thus for the Vanilla meta-interpreter, as long as all the relevant concepts are defined with respect to the underlying ambivalent language.

2.2 Correctness of the Vanilla meta-interpreter

In the present section and the two subsequent sections, we set out to investigate the correctness of the untyped Vanilla meta-interpreter. In particular, we will consider the meta-program with underlying ambivalent syntax allowing for atoms to occur as terms. Two different correctness results will be discussed, respectively proved: procedural correctness, which relates computed answer substitutions (via SLD(NF)-resolution) of the object program and the associated meta-program, and declarative correctness, which relates the intended interpretations (i.e., the least Herbrand models, as we will mainly concern definite object programs) of object and meta-program. The procedural correctness of the untyped Vanilla meta-interpreter will be shown to be a corollary of the procedural correctness result for the *typed* Vanilla meta-interpreter, as proven by Hill and Lloyd [HL88] (2.2.1).. For the declarative correctness (with respect to definite object object programs) we will present a direct proof in Section 2.4. Other approaches to declarative correctness are discussed in the Sections 2.2.2–2.2.3. In addition, we discuss the alternative approach via S-semantics in Section 2.2.4.

2.2.1 Procedural correctness

Hill and Lloyd [HL88] prove procedural correctness of the typed meta-interpreter for normal programs. In their approach, a functional representation of the object program is used. The procedural correctness of the untyped Vanilla meta-interpreter (Theorem 2.2.1 below) can be obtained as a corollary of the proof given in Hill and Lloyd [HL88]. To see this, the following observations suffice. The crucial step in this proof is to consider, instead of the original typed meta-program V_P, a partial evaluation V_P^* with respect to the atom $demo(x)$. More in particular, V_P^* is obtained by unfolding the atom $clause(x, y)$ in the (typed version of the) clause [M3], followed by unfolding the atom $demo(empty)$ in the resulting clauses. It can be easily shown that, for any object level goal $\leftarrow Q$, and any derivation d for $V_P^* \cup \{\leftarrow demo(Q)\}$, any variable occurring in d is of object type. As a consequence, although both V_P and V_P^* are typed, the typing plays a negligible role for the queries we are interested in ($demo(Q)$, where Q is an object level query). Thus, the procedural correctness result of Hill and Lloyd immediately translates to the untyped Vanilla meta-interpreter for normal object programs (and thus, by inclusion, for definite object programs). Similarly, the proof is insensitive to their particular choice of representation of the object level language (functional representation), and goes through, without modification, for the above approach with ambivalent syntax.

As a consequence of the above observations, we have the following result:

2.2.1. THEOREM. (Procedural correctness) *Let P be a normal program and $\leftarrow Q$ a normal goal. Let $\mathbf{V}_P$ be as in Definition 2.1.1, extended with the clause* [M$\neg$]. *Then the following hold:*

1. *θ is a computed answer substitution for $P \cup \{\leftarrow Q\}$ iff*
   θ is a computed answer substitution for $\mathbf{V}_P \cup \{\leftarrow demo(Q)\}$.

2. *$P \cup \{\leftarrow Q\}$ has a finitely failed SLDNF-tree iff $\mathbf{V}_P \cup \{\leftarrow demo(Q)\}$ has a finitely failed SLDNF-tree.*

2.2.2 Declarative correctness for normal object programs

In addition to their result on procedural correctness, Hill and Lloyd [HL88] prove declarative correctness of the typed Vanilla meta-interpreter for normal object programs. As the intended meaning of a normal program, the completion is taken. More precisely, it is shown in Hill and Lloyd [HL88] that correct answers for $comp(P) \cup \{\leftarrow Q\}$ coincide with correct answers for $comp(V_P) \cup \{\leftarrow demo(Q)\}$ for object level queries Q. Unlike the proof of procedural correctness, Hill and Lloyd's proof of declarative correctness does not generalise to the case of the *untyped* Vanilla meta-interpreter. The reason for this failure is that the sortedness of domains of models for the completion of the meta-program plays an essential role in this proof.

The difficulty in getting a satisfactory result on declarative completeness for the untyped case is illustrated by the restrictions on the results in Martens and De Schreye [MS95b]. There, the class of normal programs for which the associated Vanilla meta-interpreter is shown to be complete, is the class of stratifiable, language independent programs. The correctness result for this class of programs relates the perfect Herbrand model of an object program with (a suitable subset of) the weakly perfect Herbrand model of the associated meta-program. A functional representation of the object programs is used. The results do not depend on this particular choice of language, and also hold if ambivalent syntax is used as the underlying language for the Vanilla meta-interpreter. It appears that the restriction to stratifiable object programs in the results of Martens and De Schreye can be liberated somewhat to local stratifiability and weak stratifiability. In contrast, the restriction of language independence, which is closely linked to typing, seems to be crucial. At the same time, the restriction to language independent programs seriously limits the applicability of the result. In practice, as language independence is undecidable, the result applies to a syntactically defined subclass—the class of range restricted programs. Range restriction, however, is the exception rather than the rule among the basic logic programs; e.g., the programs *set*, *list*, and *member* are not range restricted.

Concluding, the declarative correctness result proven in Martens and De Schreye [MS95b] applies only to a seriously limited class of programs, but it seems that it defines the boundaries of what can be obtained with respect to the untyped Vanilla meta-interpreter for normal programs.

2.2.3 Declarative correctness for definite object programs

For *definite* object programs, a satisfactory declarative correctness result can be obtained in several ways. First, declarative correctness of the untyped version is a corollary of the procedural correctness Theorem 2.2.1, by the soundness and (strong) completeness of SLD-resolution. Again, this result holds both for the Vanilla meta-interpreter using ambivalent syntax and for the version using a functional representation of the object program. Second, a direct proof can be given, in which, by comparing different stages of the relevant fixpoint operators, the following relation between the least Herbrand models $\mathbf{M}_P$ en $\mathbf{M}_{V_P}$ of respectively the object program and the associated meta-program is established: For all object level atoms A, $A \in \mathbf{M}_P$ iff $demo(A) \in \mathbf{M}_{V_P}$. We will present a proof of this correctness result (Theorem 2.4.1) in Section 2.4.

Typically, as observed in the introduction, this declarative correctness result for definite object programs relates the least Herbrand model $\mathbf{M}_P$ of an object program P to the subset of the least Herbrand model $\mathbf{M}_{V_P}$ of the associated meta-program, not containing 'unrelated' atoms. In a corollary to the declarative correctness result Theorem 2.4.1, we obtain a satisfactory interpretation of unrelated atoms: the metalevel terms occurring in these atoms can be interpreted as variables ranging over the object level terms (Corollary 2.4.7).

A stronger correlation between the least Herbrand models $\mathbf{M}_P$ and $\mathbf{M}_{V_P}$ is established in Martens and De Schreye [MS95a] for the class of *language independent* definite object programs. A program is language independent if its least Herbrand model is not affected by extensions of the underlying language. More practically, the class of language independent programs can be characterised as the class of programs for which every computed answer substitution is ground. For language independent programs P, not only the above correctness result holds, but also the following property. If $demo(p(t) \in \mathbf{M}_{V_P}$, and p is a predicate occurring in P then t must be a closed term in the language $\mathcal{L}_P$ underlying P. In other words, unrelated atoms do not occur in $\mathbf{M}_{V_P}$, provided the object program P is language independent.

The real interest of the Vanilla meta-interpreter lies in extensions. Enhanced meta-interpreters are obtained from the Vanilla meta-interpreter by allowing extra arguments in the *demo*-predicate and extra body atoms in the clauses of the general part $\mathbf{V}$. The resulting program is then extended with clauses defining the new predicates. In general, enhanced meta-interpreters can not be expected to be complete w.r.t. object programs. In many applications, soundness rather than completeness is desired. Martens and De Schreye [MS95a] prove declarative soundness for enhanced meta-programs w.r.t. language independent programs. In addition, they obtain several correctness results for amalgamations of language independent object programs and the Vanilla meta-interpreter. We generalise their correctness result for the textual combination $\mathbf{V}_P + P$ of Vanilla with an object program P, to the class of *all* object programs (Section 2.5).

2.2.4 S-semantics

A different approach to proving correctness of the Vanilla meta-interpreter for definite programs is the use of S-semantics. The S-semantics was introduced by Falaschi et al. [FLMP93] to close the gap between the procedural and the declarative interpretations of definite programs. Essentially, the (unique) least S-herbrand model $\mathbf{M}_P^S$ of a program P consists, modulo renaming of variables, of those (not necessarily closed) atoms $Q(t)$ such that $t = x\theta$, where θ is the computed answer substitution for $P \cup \{\leftarrow Q(x)\}$. Analogous to the usual least Herbrand models, the least S-Herbrand models are the least fixed points of a proper version of the T-operator on subsets of the S-Herbrand base. Two properties are of interest in the present context. First, as observed in Levi and Ramundo [LR93], the S-semantics is independent of the language. Second, while the S-semantics provides a declarative way to express the procedural interpretation of a program, the least Herbrand model of a program can be computed from its S-semantics (by taking all the ground instances of its elements).

Independently, Levi and Ramundo [LR93] and Martens and De Schreye [MS95a] proved the following strong correspondence between the S-semantics of a definite object program and the S-semantics of its related Vanilla meta-interpreter: $demo(p(t)) \in \mathbf{M}^S_{V_P}$ iff t is an object level term and $p(t) \in \mathbf{M}^S_P$. As observed by Levi and Ramundo [LR93], both the procedural and the declarative correctness of the Vanilla meta-interpreter for definite programs are a consequence of this correctness theorem w.r.t. the S-semantics. In particular, it subsumes the results proved in Section 2.4. It should be remarked that the approach via S-semantics only applies to the case of definite programs, as an S-semantics for normal programs is not completely developed.

The approach via S-semantics has various interesting applications. Brogi and Turini [BT95] obtain in this context elegant equivalence proofs for the procedural and the meta-logical definition of basic composition operators for the construction of composite object programs. A binary demo-predicate is defined, the extra argument of which explicitly denotes the interpreted object program as a term built up from program constants and the composition operators. This predicate also differs from the unary demo predicate defined in the Vanilla meta-interpreter in that it represents the object clauses, so that an extra predicate *clause* is not used. The above cited correctness result for the Vanilla meta-interpreter in the contex of S-semantics is a special case of the more general results in Brogi and Turini [BT95]. Levi and Ramundo [LR93] obtain a correctness result for enhanced meta-interpreters defining inheritance mechanisms on structured object programs. In addition, they show that the linear overhead of meta-computation via the Vanilla meta-interpreter can be eliminated by specialising the meta-interpreter. In contrast, Martens and De Schreye give a counterexample, showing that in the context of S-semantics even soundness cannot be expected for the general class of enhanced meta-interpreters. They prove that language independence of the object program is a sufficient condition for soundness of enhanced versions of Vanilla.

2.3 Preliminaries on substitution

As we observed in Section 2.1, one of the problems involved in proving correctness of the Vanilla meta-interpreter $\mathbf{V}_P$ results from in the fact that the $\mathcal{L}_{V_P}$, the language underlying $\mathbf{V}_P$, is in several ways an extension of the language $\mathcal{L}_P$ underlying the object program P. First, the ambivalent syntax of $\mathcal{L}_{V_P}$ can be considerd as an extension of the standard syntax of $\mathcal{L}_P$. Second, the sets of predicates $\mathcal{R}_{V_P}$ occurring in $\mathbf{V}_P$ is a proper extension of the set of predicates $\mathcal{R}_P$ occurring in p. Similarly, $\mathcal{F}_P \subset \mathcal{F}_{V_P}$, and $\mathcal{C}_P \subset \mathcal{C}_{V_P}$. We will first formalise this notion of extension of a language.

For this purpose, we identify languages $\mathcal{L}$ with pairs (L, S), where L is a triple (R_L, F_L, C_L), indicating the non-logical constants of $\mathcal{L}$, and where S indicates the particular syntax of $\mathcal{L}$ (standard or ambivalent). We will only consider languages, which, like $\mathcal{L}_P$ and $\mathcal{L}_{V_P}$, have no logical constants—that is, languages of which the set of formulas consists of atoms only. This is not a necessary restriction, and the theory developed below can be generalised to the case of languages with connectives and quantifiers. In addition, we confine the discussion to atoms-as-terms ambivalence. Again, this is not a necessary restriction. Also, the theory below easily generalises to suit comparison of two languages with a different level of ambivalence. In the remainder of this section, if we mention ambivalent syntax, we always intend atoms-as-terms ambivalence, as in Definition 2.1.2.

2.3.1. DEFINITION. For two languages $\mathcal{L} = (L, S)$ and $\mathcal{L}' = (L', S')$, we say that $\mathcal{L}$ is *part of* $\mathcal{L}'$ ($\mathcal{L} \subseteq \mathcal{L}'$) if

1. $R_L \subseteq R_{L'}$, $F_L \subseteq F_{L'}$, and $C_L \subseteq C_{L'}$;
2. S is standard syntax and S' is ambivalent syntax. □

Typically, the underlying language of an object program P is part of the language of the associated meta-interpreter $\mathbf{V}_P$. An easy consequence of the above definition is the following:

2.3.2. LEMMA. *Let $\mathcal{L}$ and $\mathcal{L}'$ be two languages such that $\mathcal{L} \subseteq \mathcal{L}'$. Then* $\text{TERM}_{\mathcal{L}} \subseteq \text{TERM}_{\mathcal{L}'}$ *and* $\text{ATOM}_{\mathcal{L}} \subseteq \text{ATOM}_{\mathcal{L}'}$.

In the remainder of the present section, we will always suppose $\mathcal{L} \subseteq \mathcal{L}'$, as defined above.

A crucial step in the correctness proof for the Vanilla meta-interpreter which we present in Section 2.4, relies on a shift from (sub)terms of the language underlying $\mathbf{V}_P$ to terms of the language underlying an object program P. In the remainder of the present section we define and discuss the necessary transformation (Definition 2.3.3).

If one takes a closer look at the (tree-)structure of a term t of $\mathcal{L}'$, one sees that certain subterms of t are terms of $\mathcal{L}$, while others are proper $\mathcal{L}'$-terms. By substituting in an $\mathcal{L}'$-term t all of its subterms which are proper $\mathcal{L}'$-terms by variables, one can transform t into an $\mathcal{L}$-term, as follows: One can descend in the term-tree of t until one encounters a symbol s that is either a predicate, or a function symbol or a constant in $\mathcal{L}'$, but not a constant or function symbol of $\mathcal{L}$; replace the subtree starting with s with a fresh variable which does not occur in t. This procedure yields a term of $\mathcal{L}$. Moreover, this can be done in a systematic way, by replacing equal terms by equal variables. The *$\mathcal{L}'/\mathcal{L}$-abstraction*, which formalises this idea, thus transforms terms from $\mathcal{L}'$ into terms from $\mathcal{L}$ (Definition 2.3.3). Moreover, the transformation is reversible (Lemma 2.3.6).

2.3.3. DEFINITION. Let $\mathcal{L} \subseteq \mathcal{L}'$, and let t be a term of $\mathcal{L}'$. The *$\mathcal{L}'/\mathcal{L}$ -abstraction of t*, $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$, is inductively defined as follows:

$$\begin{array}{ll}
\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(x) = x & \text{if } x \text{ is a variable;}\\
\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(c) = c & \text{if } c \in C_L;\\
\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(c) = x_c & \text{if } c \in C_{L'} \setminus C_L,\\
\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(f(t_1,\dots,t_n)) = f(\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t_1),\dots,\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t_n)) & \text{if } f \in F_L;\\
\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(f(t_1,\dots,t_n)) = x_{f(t_1,\dots,t_n)} & \\
 & \text{if } f \in R_{L'} \cup (F_{L'} \setminus F_L).
\end{array}$$

Here, x_c and $x_{f(t_1,\dots,t_n)}$ are variables. □

2.3.4. EXAMPLE. Let $\mathcal{L}$ be the underlying (standard) language of an object program P such that $\mathcal{R}_P = \{p\}$, $\mathcal{F}_P = \{f(\cdot), g(\cdot,\cdot)\}$ and $\mathcal{C}_P = \{c\}$. Let $\mathcal{L}'$ be the underlying (ambivalent) language of the associated meta-interpreter $\mathbf{V}_P$. Then

$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(demo(f(c))) = x_{demo(f(c))}$
$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(f(c)) = f(c)$
$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(p(f(c))) = x_{p(f(c))}$
$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(f(empty)) = f(x_{empty})$
$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(g(demo(x),c)) = g(\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(demo(x)), \mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(c)) = g(x_{demo(x)}, c).$ □

Implicitly we have assumed that we have extended the set of variables of $\mathcal{L}$ with a set of fresh variables which are indexed by $\mathcal{L}'$-terms. This means that, while our purpose was to get a term of $\mathcal{L}$ as the result of an abstraction operation, we get a term in a language which is in fact an extension of $\mathcal{L}$. However, by renaming the fresh variables in $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$, we can get a term of $\mathcal{L}$ proper. The following lemma immediately follows from the above definition:

2.3.5. LEMMA. *Let t be a term of $\mathcal{L}$. Then the following hold:*

1. $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$ *is unique;*
2. $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$ *is, modulo renaming of variables, a term of $\mathcal{L}$.*

The abstraction operation is reversible, as the following lemma shows. We will sometimes write the application of a substitution σ to a term t as $t \cdot \sigma$, to increase readability.

2.3.6. LEMMA. *For every term t in $\mathcal{L}'$, there is an $\mathcal{L}$-substitution σ_t, such that $t = \mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t) \cdot \sigma_t$.*

PROOF: Define σ_t by induction on the term structure of t, as follows:

$$\begin{array}{ll}
\sigma_c = \emptyset & \text{if } c \in C_L;\\
\sigma_x = \emptyset & \text{if } x \text{ is a variable}\\
 & \quad \text{without an index from } \mathrm{TERM}_{\mathcal{L}'};\\
\sigma_{x_c} = \{x_c/c\} & \text{if } c \in C_{L'} \setminus C_L;\\
\sigma_{f(t_1,\dots,t_n)} = \sigma_{t_1} \cup \ldots \cup \sigma_{t_n} & \text{if } f \in F_L;\\
\sigma_{x_{f(t_1,\dots,t_n)}} = \{x_{f(t_1,\dots,t_n)}/f(t_1,\dots,t_n)\} & \text{if } f \in R_{L'} \cup (F_{L'} \setminus F_L).
\end{array}$$ □

2.3.7. EXAMPLE. Let $\mathcal{L}$ and $\mathcal{L}'$ be as in Example 2.3.4. Then

$f(demo(c)) = f(x_{demo(c)})\{x_{demo(c)}/demo(c)\}$
$g(f(demo(c)), empty) =$
$\qquad g(f(x_{demo(c)}), x_{empty})\{(x_{demo(c)}/demo(c), x_{empty}/empty\}$ □

In the correctness proof, we will use not only the concept of abstraction of terms, but also the more general concept of abstraction of a substitution. The abstraction of a substitution is obtained by applying the abstraction function to its substituents.

2.3.8. Definition. Let θ be an $\mathcal{L}$-substitution. The $\mathcal{L}'/\mathcal{L}$ *-abstraction of* θ, written as $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(\theta)$, is defined as follows:

$$\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(\theta) = \{x/\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t) : x/t \in \theta\}.$$

The following lemma characterises the connection between the $\mathcal{L}'/\mathcal{L}$ -abstraction of an $\mathcal{L}'$-ground term t and $\mathcal{L}$-terms s of which t is an instantiation.

2.3.9. Lemma. *Let s be a term of $\mathcal{L}$, t a term of $\mathcal{L}'$, and θ an $\mathcal{L}'$-substitution such that $s \cdot \theta = t$. Then $s \cdot \mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(\theta) = \mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$.*

Lemma 2.3.9 can be read as follows, writing θ^* for $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(\theta)$ and t^* for $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t)$: $s \cdot \theta^* = (s \cdot \theta)^*$.

2.3.10. Example. Let again $\mathcal{L}$ and $\mathcal{L}'$ be as in Example 2.3.4. Then

$$\begin{array}{lcl} g(f(empty), demo(c)) & = & g(f(x), y)\{x/empty, y/demo(c)\} \text{ and} \\ g(f(x_{empty}), x_{demo(c)}) & = & g(f(x), y)\{x/x_{empty}, y/x_{demo(c)}\} \end{array}$$ □

We extend the usual definition of ground substitution in the following way:

2.3.11. Definition. *A substitution σ is $\mathcal{L}$-ground if $\sigma = \{x_i/t_i : i \in [1, n]\}$, where t_i is a ground term of $\mathcal{L}$.*

The following lemma is an application of a well-known fact about ground substitutions.

2.3.12. Lemma. *Let t be a ground term of $\mathcal{L}'$. Let σ be an $\mathcal{L}$-ground substitution such that $dom(\sigma) \supseteq \mathrm{FV}(\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t))$. Then $\mathsf{abs}_{\mathcal{L}'/\mathcal{L}}(t) \cdot \sigma$ is a ground term of $\mathcal{L}$.*

2.4 Declarative correctness

The present section is devoted to a proof of the declarative correctness of the untyped Vanilla meta-interpreter using ambivalent syntax. This is basically a result about the relation between the least Herbrand model of a definite program and the least Herbrand model of its associated meta-program, and the proof we present proceeds by comparing stages of the respective fixpoint-operators. Hence we will be mainly concerned with ground versions of programs. As we remarked in Section 2.1, variables occurring in the object-program specific part of the Vanilla interpreter, the clauses [M4] and [M5], can be instantiated with terms from the meta-language. Therefore, we need the following extension of the usual concept of the ground version of a program.

2.4.1. Definition. Let P be a definite program. Let $\mathcal{L}$ be a language containing $\mathcal{L}_P$. With *$\mathcal{L}$-ground(P)* we mean the set of all clauses that are $\mathcal{L}$-ground instantiations of clauses of P. The natural ground version of P, *$\mathcal{L}_P$-ground(P)*, will be indicated simply as *ground(P)*. □

The next lemma is an immediate application of the Lemmas 2.3.9 and 2.3.12 to clauses of definite programs. It establishes some simple relations between the natural ground version *ground*(P) of P and the ground version obtained by instantiation with terms from the meta-language, $\mathcal{L}_{V_P}$-*ground*(P). We will speak about P-ground and $\mathbf{V}_P$-ground substitutions to refer to $\mathcal{L}_P$-ground and $\mathcal{L}_{V_P}$-ground substitutions, respectively. We will also leave out indices of the abstraction operator, that is, we will write $\mathsf{abs}(\cdot)$ instead of $\mathsf{abs}_{\mathcal{L}_{V_P}/\mathcal{L}_P}$.

2.4.2. LEMMA. *Let* $C = p(s) \leftarrow p_1(s_1), \ldots, p_n(s_n)$ *be a clause of* P.
Let $C' = p(t) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ *be in* $\mathbf{V}_P$-*ground*(P), *and let* θ *be an* $\mathbf{V}_P$-*ground substitution with* $dom(\theta) \supseteq \text{FV}(C)$, *such that* $C \cdot \theta = C'$. *Then the following hold:*

1. $C \cdot \mathsf{abs}(\theta) = p(\mathsf{abs}(t)) \leftarrow p_1(\mathsf{abs}(t_1)), \ldots, p_n(\mathsf{abs}(t_n))$;
2. *for all* $\mathbf{V}_P$-*ground substitutions* σ *with* $dom(\sigma) \supseteq ran(\mathsf{abs}(\theta))$,
   $C \cdot \mathsf{abs}(\theta) \cdot \sigma$ *is in* $\mathbf{V}_P$-*ground*(P);
3. *for all* P-*ground substitutions* σ *with* $dom(\sigma) \supseteq ran(\mathsf{abs}(\theta))$,
   $C \cdot \mathsf{abs}(\theta) \cdot \sigma$ *is in ground*(P).

PROOF: (1) immediately follows from Lemma 2.3.9. (2) follows from the definitions, and (3) follows from (1) and Lemma 2.3.12. □

We will first establish some easy facts about $\mathbf{M}_{V_P}$, the least Herbrand model of $\mathbf{V}_P$.

2.4.3. LEMMA. *If* $demo(p(t_1, \ldots, t_n)) \in \mathbf{M}_{V_P}$, *where* p *is not* & *and* p *is not empty, then* $p \in \mathcal{R}_P$.

PROOF: Suppose p is not & or *empty*, and $demo(p(t_1, \ldots, t_n)) \in \mathbf{M}_{V_P}$. Then there is a d such that $demo(p(t_1, \ldots, t_n)) \in \mathrm{T}_{V_P}\!\uparrow\! d \setminus \mathrm{T}_{V_P}\!\uparrow\!(d-1)$. Then $demo(p(t_1, \ldots, t_n))$ must have entered by an application of the meta-clause [M3]. Therefore, for some E, $\{clause(p(t_1, \ldots, t_n), E),\ demo(E)\} \subseteq \mathrm{T}_{V_P}\!\uparrow\!(d-1)$. Thus $p(t_1, \ldots, t_n) \leftarrow E$ (or, in the case that E is the constant *empty*, $p(t_1, \ldots, t_n) \leftarrow$) belongs to $\mathbf{V}_P$-*ground*(P). So p must be a predicate in the underlying language of P. □

An immediate consequence of this lemma is that atoms of the form $demo(demo(t))$ and $demo(clause(s, t))$ do not occur in $\mathbf{M}_{V_P}$, under the reasonable assumption that $\{demo(\cdot), clause(\cdot, \cdot)\} \cap \mathcal{R}_P = \emptyset$. By a similar argument one sees that:

2.4.4. LEMMA. *Let* $\& \notin \mathcal{R}_P$ *and* $demo(A\&B) \in T_{V_P}\!\uparrow\! n$, *for some* n. *Then there are* $m, k < n$ *such that* $demo(A) \in T_{V_P}\!\uparrow\! m$ *and* $demo(B) \in T_{V_P}\!\uparrow\! k$.

PROOF: Under the assumption that $\& \notin \mathcal{R}_P$, the only clause in $\mathbf{V}_P$ of which the head unifies with $demo(A\&B)$ is [M2]. □

2.4.5. COROLLARY. *Let* $\& \notin \mathcal{R}_P$ *and* $demo(A_1\& \ldots \&A_k) \in T_{V_P}\!\uparrow\! n$ *for some* n. *Then, for* $i \in [1, k-1]$, $demo(A_i) \in T_{V_P}\!\uparrow\!(n-i)$, *and* $demo(A_k) \in T_{V_P}\!\uparrow\!(n-k+1)$.

In the remainder of this section, we will abstract from arities of predicates. That is, we will only consider unary (object level) predicates. This simplifies notation, while it does not affect the generality of proofs and results.

The next lemma is crucial for the proof of the soundness part of the correctness theorem. It expresses the main idea behind the soundness proof.

2.4.6. LEMMA. *Let P be a program such that & and empty $\notin \mathcal{R}_P$. Let $p \in \mathcal{R}_P$, and let t be a closed term in $\mathcal{L}_{V_P}$. Suppose demo$(p(t)) \in T_{V_P}\uparrow n$, for some n. Then, for all $\mathbf{V}_P$-ground substitutions σ such that dom$(\sigma) \supseteq$ FV(abs(t)), demo$(p(\text{abs}(t)\cdot\sigma)) \in T_{V_P}\uparrow n$.*

PROOF: By induction on the stages of the T_{V_P}-operator.
Let P, t, p, and σ be as in the formulation of the lemma. The lemma trivially holds for $n < 2$. Let, for some $n \geq 2$, $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow n$, and suppose that the lemma is true for all $m < n$. Then, by definition of the T_{V_P}-operator, there are $p_1, \ldots, p_k \in \mathcal{R}_P$, $\mathcal{L}_P$-terms $s, s_1, \ldots, s_k$, and an $\mathbf{V}_P$-ground substitution θ such that
(1) $p(s) \leftarrow p_1(s_1), \ldots, p_k(s_k) \in P$;
(2) $(p(s) \leftarrow p_1(s_1), \ldots, p_k(s_k)) \cdot \theta = p(t) \leftarrow p_1(t_1), \ldots, p_k(t_k)$;
(3) $clause(p(t), p_1(t_1)\& \ldots \&p_k(t_k)) \in \mathrm{T}_{V_P}\uparrow(n-1)$;
(4) $demo(p_1(t_1)\& \ldots \&p_k(t_k)) \in \mathrm{T}_{V_P}\uparrow(n-1)$.
(For $n = 2$, the situation is slightly, but not essentially, different: the P-clause in (1) has an empty body.)
Now let τ be a P-ground substitution such that
$dom(\tau) \supseteq$ FV(abs(θ)) and $\tau_{|\mathrm{FV}(\text{abs}(t))} = \sigma$.
By (1), (2), Lemma 2.4.2, and the definition of $\mathbf{V}_P$, we see that
(a) $clause(p(\text{abs}(t)\tau), p_1(\text{abs}(t_1)\tau)\& \ldots \&p_n(\text{abs}(t_n)\tau)) \in \mathrm{T}_{V_P}\uparrow(n-1)$.
From (4) and Corollary 2.4.5, it follows that $demo(p_i(t_i) \in \mathrm{T}_{V_P}\uparrow(n-1-i)$ for $i \in [1, k-1]$, and $demo(p_k(t_k) \in \mathrm{T}_{V_P}\uparrow(n-k)$. Thus, by inductive hypotheses, we have that $demo(p_i(\text{abs}(t_i)\tau)) \in \mathrm{T}_{V_P}\uparrow(n-1-i)$ for $i \in [1, k-1]$, and $demo(p_k(\text{abs}(t_k)\tau)) \in \mathrm{T}_{V_P}\uparrow(n-k)$. By $k-1$ applications of the meta-clause for conjunction [M2], it follows that
(b) $demo(p_1(\text{abs}(t_1)\tau)\& \ldots \&p_k(\text{abs}(t_k)\tau)) \in \mathrm{T}_{V_P}\uparrow(n-1)$.
By definition of the T_{V_P}-operator (using [M3], (a) and (b)), we conclude that
$demo(p(\text{abs}(t)\sigma)) = demo(p(\text{abs}(t)\tau)) \in \mathrm{T}_{V_P}\uparrow n$. □

This lemma has the following obvious generalisation:

2.4.7. COROLLARY. *Let P be a program such that & and empty $\notin \mathcal{R}_P$.*
Let $p_1, \ldots, p_k \in \mathcal{R}_P$, and let $t_1, \ldots, t_k$ be terms of $\mathcal{L}_{V_P}$.
Suppose that demo$(p_1(t_1)\& \ldots \&p_k(t_k)) \in T_{V_P}\uparrow n$, for some n.
Let σ be a $\mathbf{V}_P$-ground substitution such that dom$(\sigma) \supseteq$ FV(abs(t_1))$\cup \ldots \cup$FV(abs(t_k)).
Then demo$(p_1(\text{abs}(t_1)\sigma)\& \ldots \&p_k(\text{abs}(t_k)\sigma)) \in T_{V_P}\uparrow n$.

We are now in position to prove the correctness theorem. The proof of the soundness part heavily depends on Lemma 2.4.6 and its corollary. The proof of the completeness lemma 2.4.9 is inspired by the corresponding proof in Martens and De Schreye [MS95a].

2.4.8. LEMMA. (Soundness) *Let P be a program such that & and empty $\notin \mathcal{R}_P$, and let $\mathbf{V}_P$ be the associated Vanilla meta-interpreter.*

Then, for all $p(t) \in \mathbf{B}_P$,
$\forall n \in \mathbb{N}\ [demo(p(t)) \in T_{V_P}\uparrow n \Longrightarrow \exists m \in \mathbb{N}.\ p(t) \in T_P\uparrow m]$.

PROOF: By induction on n. Let $p(t) \in \mathbf{B}_P$. The lemma trivially holds for $n < 2$, because *empty* $\notin \mathcal{R}_P$.

Suppose $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow 2$. Then there must be a P-term s and an $\mathbf{V}_P$-ground instantiation θ such that $p(s) \leftarrow\ \in P$ and $s\theta = t$. However, by the assumption that $p(t) \in \mathbf{B}_P$, (the relevant part of) θ must be a P-ground substitution. So $p(t) \leftarrow\ \in ground(P)$ and $p(t) \in \mathrm{T}_P\uparrow 1$.

Suppose that, for some $n > 2$, $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow n \setminus \mathrm{T}_{V_P}\uparrow(n-1)$. Assume that the lemma holds for all $n' < n$. Then, by the assumption that & and *empty* $\notin \mathcal{R}_P$, the $demo(p(t))$ must have entered the Herbrand model by an application of [M3]. So there must be ground atoms $p_1(t_1), \ldots, p_k(t_k) \in \mathbf{B}_{M_P}$ such that
(1) $clause(p(t), p_1(t_1)\& \ldots \& p_k(t_k)) \in \mathrm{T}_{V_P}\uparrow 1$;
(2) $demo(p_1(t_1)\& \ldots \& p_k(t_k)) \in \mathrm{T}_{V_P}\uparrow(n-1)$.
From the definition of $\mathbf{V}_P$ it follows that $p_1, \ldots, p_k \in \mathcal{R}_P$, and that there are P-terms $s_1, \ldots, s_k$, and an $\mathbf{V}_P$-ground substitution θ such that
(3) $(p(t) \leftarrow p_1(s_1), \ldots, p_k(s_k)) \cdot \theta = p(t) \leftarrow p_1(t_1), \ldots, p_k(t_k) \in \mathbf{V}_P\text{-}ground(P)$. Now take a P-ground substitution σ such that $dom(\sigma) = ran(\mathsf{abs}(\theta))$. By Lemma 2.4.2,
$p(t) \leftarrow p_1(s_1 \cdot \mathsf{abs}(\theta) \cdot \sigma), \ldots, p_n(s_n \cdot \mathsf{abs}(\theta) \cdot \sigma) \in ground(P)$. [a]
From (2), the Corollaries 2.4.5 and 2.4.7, and the inductive hypothesis, we can conclude that
$\exists m_i \in \mathbb{N}.\ p_i(\mathsf{abs}(t_i) \cdot \sigma) \in \mathrm{T}_P\uparrow m_i$, for $i \in [1, k]$. [b]
Now from [a] and [b], we can conclude that there is an m such that $p(t) \in \mathrm{T}_P\uparrow m$.□

2.4.9. LEMMA. (Completeness) *Let P be a program and let $\mathbf{V}_P$ be the associated Vanilla meta-interpreter. Then for all* $p(t) \in \mathbf{B}_P$,
$\forall n \in \mathbb{N}\ [p(t) \in T_P\uparrow n \Longrightarrow \exists m \in \mathbb{N}.\ demo(p(t)) \in T_{V_P}\uparrow m]$.

PROOF: By course of values induction on n. The lemma trivially holds for $n = 0$.

Suppose $p(t) \in \mathrm{T}_P\uparrow 1$. Then $clause(p(t), empty) \leftarrow\ \in ground(\mathbf{V}_P)$ and therefore, $clause(p(t), empty) \in \mathrm{T}_{V_P}\uparrow 1$. Also, $demo(empty) \in \mathrm{T}_{V_P}\uparrow 1$, so, by [M3], $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow 2$. Suppose that $p(t) \in \mathrm{T}_P\uparrow n \setminus \mathrm{T}_P\uparrow(n-1)$, for some $n > 1$, and suppose that the lemma holds for all $n' < n$. Then there are $p_1(s_1), \ldots p_k(s_k) \in \mathbf{B}_P$, such that $p(t) \leftarrow p_1(s_1), \ldots p_k(s_k) \in ground(P)$, and $p_i(s_i) \in \mathrm{T}_P\uparrow(n-1)$ for $i \in [1, k]$. Thus, by the inductive hypothesis, there are m_i such that $demo(p_i(s_i)) \in \mathrm{T}_{V_P}\uparrow m_i$, for $i \in [1, k]$. Thus, by repeated use of [M2], there is an m such that $demo(p_1(s_1)\& \ldots \& p_k(s_k)) \in \mathrm{T}_{V_P}\uparrow m$. Also, $clause(p(t), p_1(s_1)\& \ldots \& p_k(s_k)) \leftarrow\ \in ground(\mathbf{V}_P)$. As a consequence, $clause(p(t), p_1(s_1)\& \ldots \& p_k(s_k)) \in \mathrm{T}_{V_P}\uparrow m$.
We conclude that $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow(m+1)$. □

2.4.1. THEOREM. (Correctness) *Let P be a program such that & and empty $\notin \mathcal{R}_P$, and let $\mathbf{V}_P$ be the Vanilla meta-interpreter associated with P. Then for all* $p(t) \in \mathbf{B}_P$,

$$p(t) \in \mathbf{M}_P \text{ iff } demo(p(t)) \in \mathbf{M}_{V_P}.$$

As an immediate consequence of the correctness theorem and Lemma 2.4.6 is the following corollary:

2.4.10. COROLLARY. *Let $demo(p(t)) \in \mathbf{M}_{V_P}$, where p is a predicate in $\mathcal{L}_P$. Let σ be an $\mathcal{L}_P$-ground substitution with $dom(\sigma) = var(\mathsf{abs}(t))$. Then $p(\mathsf{abs}(t) \cdot \sigma) \in \mathbf{M}_P$.*

The above corollary shows that the occurrence of unrelated atoms in the least Herbrand model of the Vanilla meta-program is less of a problem than usually thought: the meta-level terms occurring in unrelated atoms can be interpreted as free variables, ranging over the object level terms.

It should be observed that, by Lemma 2.4.3, the proof of the correctness theorem above is also a proof of the correctness theorem for the Vanilla meta-interpreter with a functional representation of the object program. The above corollary likewise translates into the functional case.

Observe that, unlike for completeness, there are some syntactical conditions on the object program in the soundness lemma, which ensure a certain degree of disjointness of the language of the object program and the Vanilla meta-interpreter. Violation of these coditions may destroy soundness of the Vanilla meta-interpreter.

First, observe that soundness is not guaranteed if & occurs as a predicate in the object program.

2.4.2. COUNTEREXAMPLE. Consider the following object program P.

$$\begin{aligned}&P(a) \leftarrow \\ &Q(a) \leftarrow P(f(a)), P(a) \\ &x\&y \leftarrow\end{aligned}$$

Now $Q(a) \notin \mathbf{M}_P$. In contrast, $demo(P(f(a))\&P(a)) \in \mathbf{M}_{\mathbf{V}_P}$, so $demo(Q(a)) \in \mathbf{M}_{\mathbf{V}_P}$. Thus $\mathbf{V}_P$ is not declaratively sound w.r.t. P. □

Similarly, soundness is not guaranteed if *empty* occurs as an atom in the language of the object program.

2.4.3. COUNTEREXAMPLE. Consider the following object program Q.

$$A \leftarrow empty$$

Now $A \notin \mathbf{M}_P$, while $demo(A) \in \mathbf{M}_{V_P}$. □

The above correctness theorem should be compared with the corresponding result for the class of language independent programs (Theorem 13 of Martens and De Schreye [MS95a].) The latter is stronger, in the sense that it additionally shows that for language independent programs, no unrelated atoms can occur in the least Herbrand model of the meta-program.

In the above, we have silently assumed that the language underlying the object program has a non-empty set of constants $\mathcal{C}_P$. Adopting the usual convention that, in case $\mathcal{C}_P$ is empty, the Herbrand universe is built up using a generic constant $*$, the following special case of the correctness theorem holds: for all $p \in R_{\mathcal{L}_P}$,

$$p(*) \in \mathbf{M}_P \text{ iff, for all } t \in \text{CLTERM}_{\mathcal{L}_P}, demo(p(t))\mathbf{M}_{V_P}.$$

This follows easily from adaptations of the proofs of the lemma's 2.4.8 and 2.4.9.

The above completeness proof provides evidence of the linear overhead of meta-level computation. In particular, an explicit measure for this overhead (on the declarative level) can be extracted from the proof. We need the following definitions.

2.4.11. DEFINITION. *Let P be a program consisting of the clauses $C_1, \ldots, C_k$.*
The body depth *of $d(C)$ of a clause C is the number of its body atoms, i.e.,*
$d(C) = n$ if C is a clause $A \leftarrow B_1, \ldots, B_n$, and
$d(C) = 0$ if C is a fact clause $A \leftarrow$.
The body depth *d_P of P is the maximum of the depths of the clauses of P, i.e.,*
$d_P = max\{d(C_i) : i \in [1,k]\}$. □

2.4.12. PROPOSITION. *Let P be a program.*
For $n > 0$, if $p(t) \in T_P\uparrow n$, then $demo(p(t)) \in T_{V_P}\uparrow(2 + d_P(n-1))$.

PROOF: The proof proceeds by induction, and follows the proof of Lemma 2.4.9. Clearly, if $p(t) \in \mathrm{T}_P\uparrow 1$, the $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow 2$, so the proposition holds for $n = 1$. Let $n > 1$, and suppose the proposition holds for $n - 1$.
Let $p(t) \in \mathrm{T}_P\uparrow n \setminus \mathrm{T}_P\uparrow(n-1)$. Then there are $p_1(s_1), \ldots p_k(s_k) \in \mathbf{B}_P$, such that $p(t) \leftarrow p_1(s_1), \ldots p_k(s_k) \in ground(P)$, and $p_i(s_i) \in \mathrm{T}_P\uparrow(n-1)$ for $i \in [1,k]$. By the inductive hypothesis, $demo(p_i(s_i)) \in \mathrm{T}_{V_P}\uparrow(2 + d_P(n-2))$, for $i \in [1,k]$. Thus, by $k-1$ applications of [M3], $demo(p_1(s_1)\&\ldots\&p_k(s_k)) \in \mathrm{T}_{V_P}\uparrow(2 + d_P(n-2) + k - 1)$. Then, by an application of [M2], $demo(p(t)) \in \mathrm{T}_{V_P}\uparrow(2 + d_P(n-2) + k)$. Observe that $k \leq d_P$. The proposition now follows from the monotonicity of the fixpoint operator. □

A similar linear relation between the length of object-level and meta-level SLD-derivations can be established. We refer for this result to Levi and Ramundo [LR93].

2.5 An amalgamation

The correctness of the Vanilla meta-interpreter in fact shows that it proves no more nor less than the object program. It is extensions of the Vanilla meta-interpreter where the real interest lies. The theory developed in the present chapter can be used to obtain correctness results for those extensions. We mention the following preliminary result on a simple amalgamation, the textual combination $\mathbf{V}_P \cup P$ of $\mathbf{V}_P$ with P.

2.5.1. THEOREM. *Let P be a program such that*
$\{(\cdot)\&(\cdot), empty, demo(\cdot), clause(\cdot,\cdot)\} \cap \mathcal{R}_P = \emptyset$. Then
for all (ground) A in $\mathcal{L}_{V_P}$, $demo(A) \in \mathbf{M}_{V_P}$ iff $demo(A) \in \mathbf{M}_{V_P \cup P}$;
for all (ground) A in $\mathcal{L}_P$, $A \in \mathbf{M}_P$ iff $A \in \mathbf{M}_{V_P \cup P}$.

PROOF: Along the same lines as the proof of Theorem 2.4.1, while using the following observations: The underlying language of $\mathbf{V}_P \cup P$ is $\mathcal{L}_{V_P}$; Clauses of P can now be instantiated with ground terms from $\mathcal{L}_{V_P}$; Lemma 2.4.6 also holds for atoms $p(t) \in \mathbf{M}_{V_P}$ for which $p \in \mathcal{R}_P$; By the assumptions on P, atoms from the bodies of clauses of $\mathbf{V}_P$ do not unify with heads of clauses from P—and the converse. □

Alternatively, another proof of the above theorem is obtained by examining properties of the fixpoint operator $T_{V_P \cup P}$ of the amalgamated program. As observed in Brogi and Turini [BT95], it is in general not true that the least Herbrand model of the textual combination $P \cup Q$ of two programs P and Q is the union of their respective least Herbrand models. However, for certain pairs of programs, the least Herbrand model of their union does equal the union of the respective least Herbrand models w.r.t. the union of the languages. We will use the following notion:

2.5.1. DEFINITION. *Two programs P and Q are* non-connected *if*
(i) none of the atoms occurring in the bodies of clauses of $\mathcal{L}_{P \cup Q} - ground(P)$ occurs as the head of a clause in $\mathcal{L}_{P \cup Q} - ground(Q)$, and
(ii) none of the atoms occurring in the bodies of clauses of $\mathcal{L}_{P \cup Q} - ground(Q)$ occurs as the head of a clause in $\mathcal{L}_{P \cup Q} - ground(P)$. □

We need the following generalisation of the notion of T_P-operator and its fixed point.

2.5.2. DEFINITION. *Let P be a program, and let $\mathcal{L} \supseteq \mathcal{L}_P$.*
$T_{P,\mathcal{L}}(I) := \{A : A \leftarrow B_1, \ldots, B_n \in \mathcal{L} - ground(P) \text{ and } \{B_1, \ldots, B_n\} \subseteq I\}$
$T^{\omega}_{P,\mathcal{L}}(\emptyset) := \bigcup T^{n}_{P,\mathcal{L}}(\emptyset)$. □

In addition, we can, as usual, identify the least $\mathcal{L}$-Herbrand model $\mathbf{M}^{\mathcal{L}}_P$ with $\mathrm{T}^{\omega}_{P,\mathcal{L}}(\emptyset)$.

2.5.3. LEMMA. *Let P and Q be two non-connected programs, and let $\mathcal{L}$ be the union of their respective underlying languages. Then $\mathbf{M}_{P \cup Q} = \mathbf{M}^{\mathcal{L}}_P \cup \mathbf{M}^{\mathcal{L}}_Q$.*

PROOF: By induction on the stages of the $\mathrm{T}_{P \cup Q}$-operator.
$\mathrm{T}_{P \cup Q}\uparrow 0 = \mathrm{T}_{P,\mathcal{L}}\uparrow 0 \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow 0 = \emptyset$, by definition.
Suppose that for some n, $\mathrm{T}_{P \cup Q}\uparrow n = \mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n$. Then

$\mathrm{T}_{P \cup Q}\uparrow n + 1 =$
{ by definition}
$= \mathrm{T}_{P \cup Q}(\mathrm{T}_{P \cup Q}\uparrow n)$
{by the inductive hypothesis}
$= \mathrm{T}_{P \cup Q}(\mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n)$
{ by definition of $\mathrm{T}_{P \cup Q}$}
$= \{A : A \leftarrow B_1, \ldots, B_k \in ground(P \cup Q) \& \{B_1, \ldots, B_k\} \subseteq \mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n\}$
{because $ground(P \cup Q) = \mathcal{L} - ground(P) \cup \mathcal{L} - ground(Q)$}
$= \{A : A \leftarrow B_1, \ldots, B_k \in \mathcal{L} - ground(P) \& \{B_1, \ldots, B_k\} \subseteq \mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n\}$
$\quad \cup \{A : A \leftarrow B_1, \ldots, B_k \in \mathcal{L} - ground(Q) \& \{B_1, \ldots, B_k\} \subseteq \mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n\}$
{because P and Q are non-connected }
$= \{A : A \leftarrow B_1, \ldots, B_k \in \mathcal{L} - ground(P) \& \{B_1, \ldots, B_k\} \subseteq \mathrm{T}_{P,\mathcal{L}}\uparrow n\}$
$\quad \cup \{A : A \leftarrow B_1, \ldots, B_k \in \mathcal{L} - ground(Q) \& \{B_1, \ldots, B_k\} \subseteq \mathrm{T}_{Q,\mathcal{L}}\uparrow n\}$
{by definition of $\mathrm{T}_{P,\mathcal{L}}\uparrow n + 1$ and $\mathrm{T}_{Q,\mathcal{L}}\uparrow n + 1$}
$= \mathrm{T}_{P,\mathcal{L}}\uparrow (n + 1) \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow (n + 1)$.

Thus,
$\mathbf{M}_{P \cup Q} = \mathrm{T}^{\omega}_{P \cup Q} = \bigcup \mathrm{T}_{P \cup Q}\uparrow n =$

$= \bigcup(\mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \mathrm{T}_{Q,\mathcal{L}}\uparrow n) = \bigcup \mathrm{T}_{P,\mathcal{L}}\uparrow n \cup \bigcup \mathrm{T}_{Q,\mathcal{L}}\uparrow n = \mathrm{T}^{\omega}_{P,\mathcal{L}} \cup \mathrm{T}^{\omega}_{Q,\mathcal{L}} = \mathbf{M}^{\mathcal{L}}_{P} \cup \mathbf{M}^{\mathcal{L}}_{Q}$. □

The above general lemma in particular applies to the amalgation of object and meta-program.

2.5.2. THEOREM. *Let P be a program such that* $\{(\cdot)\&(\cdot), \mathit{empty}, \mathit{demo}(\cdot), \mathit{clause}(\cdot,\cdot)\} \cap \mathcal{R}_P = \emptyset$. *Then* $\mathbf{M}_{V_P \cup P} = \mathbf{M}_{V_P} \cup \mathbf{M}_P^{\mathcal{L}_{V_P}}$.

PROOF: We leave if to the reader to check that $\mathbf{V}_P$ and P are indeed non-connected. (Essential here is the assumption on the language of the object program.) By the fact that $\mathcal{L}_{V_P} \supseteq \mathcal{L}_P$, the theorem is now immediately follows from the above lemma. □

As an immediate consequence of the above theorem and the completeness theorem 2.4.1, we have the following corollary.

2.5.4. COROLLARY. *Let P be a program such that* $\{(\cdot)\&(\cdot), \mathit{empty}, \mathit{demo}(\cdot), \mathit{clause}(\cdot,\cdot)\} \cap \mathcal{R}_P = \emptyset$. *Let A be a closed term of $\mathcal{L}_P$. The following are equivalent:*

(i) $\mathit{demo}(A) \in \mathbf{M}_{V_P \cup P}$
(ii) $\mathit{demo}(A) \in \mathbf{M}_{V_P}$
(iii) $A \in \mathbf{M}_P$
(iv) $A \in \mathbf{M}_{V_P \cup P}$.

Other examples involving amalgamated and non-amalgamated extensions of the Vanilla meta-interpreter can be found in Martens and De Schreye [MS95a]. A theorem similar to the above (Theorem 15) is proven there for language independent object programs P.

2.6 Conclusions

In the present chapter we have studied correctness of the untyped Vanilla meta-interpreter in the context of ambivalent syntax. Ambivalent syntax, characterised by the occurrence of atoms as terms, is the proper underlying syntax of amalgamated extensions of the Vanilla meta-interpreter.

We have shown that the proof of procedural correctness for the typed version of the Vanilla meta-interpreter for normal programs has great generality, and implies correctness for the (standard) untyped Vanilla meta-interpreter for normal programs. In contrast, the declarative correctness result for the typed Vanilla meta-interpreter does not translate into declarative correctness for the untyped version(s), due to the essential role of types. Declarative correctness for the untyped Vanilla meta-interpreter has been proven for a limited class of normal programs, which excludes most of the basic Prolog programs. For definite object programs however, a satisfying declarative correctness result can be obtained in two ways. By the usual soundness and completeness of SLD-resolution, declarative correctness for definite object programs is a consequence of the procedural correctness. We have presented a direct proof of declarative correctness for the Vanilla meta-interpreter for definite object programs with ambivalent syntax as the underlying language. The proof generalises to a corresponding proof of the correctness of the variant of the Vanilla meta-interpreter that uses a functional representation of the object program. Finally, we have presented two different proofs of the correctness of a simple amalgamation of object and meta-program.

The interest of the reported results is two-fold. First, we have shown that various correctness results (procedural correctness of the typed interpreter and declarative correctness of the untyped interpreter using ambivalent syntax) have a great generality. Second, we have shown that ambivalent syntax is useful and appropriate in the context of meta-programming.

The strong similarities between the various correctness proofs discussed in this chapter, suggest further research, investigating necessary and sufficient conditions for soundness and completeness of Vanilla style demo-predicates. An open question is whether the correctness results discussed are affected in the case of object programs with ambivalent syntax, in particular with variables occurring as atoms in bodies. In addition, a further investigation of sufficient conditions for soundness and completeness of extended meta-interpreters may be of some practical importance.

Note

This chapter is a slightly modified version of [Kal95a], "Correctness of the Vanilla meta-interpreter and Ambivalent Syntax", in: Meta-Logics and Logic Programming, Eds. K.R. Apt and F. Turini, The MIT Press, 1995, pp. 1–26.

Part III

Gentzen Systems for Logic Programming Styles

Introduction

While the classical soundness and completeness results for Logic Programming establish a strong relation between the declarative (intended) and the procedural interpretations of programs, the effects of the computation mechanism in implementations of Logic Programming such as Prolog, are not accounted for by these results. That is, in defiance of the Logic Programming theory, the procedural meaning of an implemented program need not coincide with its declarative meaning. Two examples serve to illustrate the effect of the standard Prolog computation mechanism, processing clauses in top-down order and processing goals in left-to-right order:

- The goal A succeeds under SLDNF resolution from the program which consists of the two clauses (1) $A \leftarrow A, B$ and (2) $A \leftarrow$. Under Prolog computation however, the search for A will go into a loop via clause (1).
- Consider the program consisting of the single clause $A \leftarrow B, A$. The goal $notA$ succeeds from this program under Prolog computation with the standard leftmost selection rule. In contrast, if a rightmost selection were used, the search for $notA$ would go into a loop.

This illustrates that both the search rule of a computation mechanism (the order in which the clauses are processed) and the selection rule (the order in which composite goals are processed) affect computational outcome. The objective of the subsequent chapters is to capture the effects of fixed computation mechanisms in the format of Gentzen style sequent calculi. The calculi we will discuss have as characteristic expressions sequents $[P] \Rightarrow A$, expressing success of A from the program P under the intended computation style.

Typically, computation on a logic program with only definite clauses (that is, not involving negation) via a fixed computation mechanism corresponds to a logic weaker than classical logic. Reflecting this, (most of) the procedural calculi we study below, omit or modify certain classical structural rules. Weakening of structural rules in sequent calculi results in so-called 'substructural' logics. Well-known computational examples of substructural systems are linear logics and nonmonotonic logics. The following observations illustrate the fact that procedural aspects of Logic Programming can be successfully described in the context of substructural logic:
Unrestricted addition of clauses to a program can destroy successful search. For instance, extension of a program P with a clause $A \leftarrow A$ need not preserve the success of a goal A. Some cautious forms of addition on the other hand preserve derivability. As an example, let P, R, and Z be Horn clause programs, and let C be a Horn clause. Using top-down clause processing, if a goal A succeeds from the program $P; C; R; Z$ then it also succeeds from $P; C; R; C; Z$, and vice versa. Thus top-down clause processing—as used in Prolog computation—obeys structural rules from sequent calculus like rightward extension and rightward contraction, while full weakening (addition of clauses to a program) is not valid.

We will show for several computation mechanisms for Logic Programming that they can be captured by suitable 'substructural' calculi in the Gentzen format. In particular, we investigate depth first search, corresponding to top-down clause processing. To avoid the extra complication of dealing with unification, we restrict our attention to the propositional case.

Our approach, using Gentzen style sequent calculi for the characterisation of Logic Programming styles, originates in a paper by van Benthem [vB92], who proposed a Gentzen style calculus for top-down clause processing combined with parallel goal processing. We refine some of the results reported there. In particular, we provide a formal computational semantics for the calculus proposed by van Benthem and formalise the intuitive arguments for soundness and completeness. Also, we show that several of the rules proposed in [vB92] are redundant. We will pay some attention to proof theoretic properties of procedural calculi we define. We will also study the effect of the addition of the classical structural rules and the cut rule. Finally, we will show how negation as failure can be incorporated in the various calculi in a natural way.

Next, we show how the calculus for parallel goal selection can be extended with introduction rules defining procedural versions of the usual connectives for disjunction and conjunction. These connectives have a natural interpretation in the procedural context: they effect local changes of the overall selection rule. In particular, restricting our consideration to one of these procedural connectives, we propose a variant of the usual Prolog computation procedure (frugal Prolog), which has less extensive backtracking, and therefore travels through the searchspace faster than Prolog. This alternative computation will be shown to be sound and complete w.r.t. standard Prolog computation on a large class of propositional programs.

In Chapter 4 we focus on Gentzen style axiomatisations of standard Prolog search, with top down clause processing and left-most goal selection. Due to Prolog's extensive backtracking, a correct axiomatisation of standard Prolog search requires an extension of the Gentzen format appropriate for the more simple alternatives, with sequents $[P] \Rightarrow^* A$ expressing success of A and safety (non-divergence) during backtracking. In Chapter 5, we show how the standard Prolog cut, allowing for control of the search space, can be incorporated in the calculus for Prolog by the addition of two rules reflecting the pruning behaviour of activated cuts. Variants of the standard cut will also be discussed.

For all of the discussed calculi, correctness results will be proved using as a straightforward procedural semantics relevant parts of the search trees appropriate for the underlying search procedure.

The approach taken here differs from axiomatisations of the Prolog computation procedure known from the literature, by its slightly dynamic nature. The alternative approaches [Stä94], [Stä], [And93], [Cer92] are axiomatic in the sense that they involve translations of programs into appropriate theories. The approach taken in ([Min90] and [She92], where in the course of a derivation an appropriate search tree is built up, likewise involves derivations on a single program at a time. In contrast, in the Gentzen calculi we discuss, programs are built up in the course of a derivation, and derivations do not involve any translation of programs into theories.

Notation and conventions

- A, B, C, etc. (sometimes indexed) will indicate propositional atoms.
- $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, etc. will indicate clauses.
- P, Q, R, Z, etc. will indicate programs.
- $P; R$ will indicate the concatenation of two programs P and R.
- $\emptyset$ will indicate the empty program.
- We extend the propositional language with a propositional atom $\top$ (verum), which can be understood procedurally as an atom that by definition succeeds immediately from every program. Alternatively, it can be interpreted as the empty goal.
- An atom A will be called a *proper atom* if it is not $\top$.
- $\mathcal{A}$ is a *(definite) clause* if $\mathcal{A}$ is $A \leftarrow B_1, \ldots, B_n$, where A is a proper atom and the B_i are atoms. In particular, $\top$ does not occur as the head of any clause.
- Clauses of the form $A \leftarrow \top$ take the role of the usual fact-clauses $A \leftarrow$, and will sometimes be notated as A .
- P is a *definite program* if P is a finite list of definite clauses.
- We will use set inclusion $P \subseteq R$ for two programs P and R to indicate that every clause of P occurs in R, *regardless* of the order of the clauses.

Chapter 3

Depth First Search

In the present chapter we will investigate the following computation mechanism, proposed in [vB92]:

> **General Depth First Search**
> To prove a goal A from a program, first try the uppermost program clause which has A for its head. The goal A succeeds via this clause if all its body atoms recursively succeed. The goal A fails via this clause if some body atom fails in finitely many steps, and none of the searches for the body atoms gets stuck in a loop. In the latter case, the next lower eligible program clause is tried. A fails from P if it fails consecutively via all the clauses with head A. A succeeds from P if it succeeds via an A-clause after failing via earlier A-clauses.

The general depth-first search inference mechanism is characterised by Prolog search (depth-first) with a generalised selection rule, which we will refer to as parallel selection. Parallel selection leads us to think of program clauses as being of the form $A \leftarrow \{B_1, \ldots, B_n\}$; that is, the antecedents form a set, rather than a list. Note that parallel selection is not a selection rule in the strict sense (cf. [Llo87] and [Apt90]). Rather, it can be thought of as an unspecified selection rule: if, given a clause $A \leftarrow B_1, \ldots, B_n$ and an arbitrary distribution of success and (finite) failure among the body atoms B_i, the goal A succeeds (respectively fails) via this clause under parallel selection, then it will also succeed (respectively fail) under any specific selection rule. Due to the generality of the parallel selection rule, the above inference mechanism provides a good basis for the development of a general framework for the axiomatisation of success and failure under various inference mechanisms.

3.1 A Gentzen Calculus for Depth-First Search

We will axiomatise the effects of general depth first search using the format of a Gentzen style sequent calculus **S**. The characteristic expressions of **S** are sequents of the form $[P] \Rightarrow B$, where P is a (propositional, Horn clause) program and B is an atom. The intended interpretation is as follows: a sequent $[P] \Rightarrow B$ is derivable in the calculus **S** if and only if B succeeds from P under general depth-first search.

The deductive calculus **S** consists of the following axioms and rules:

S

Axioms

$$[P] \Rightarrow \top$$

Heterogeneous Permutation

$$\frac{[P;\mathcal{C};\mathcal{D};R] \Rightarrow A}{[P;\mathcal{D};\mathcal{C};R] \Rightarrow A}\text{ HP} \qquad \text{where } \mathcal{D} \text{ and } \mathcal{C} \text{ are clauses with different heads}$$

Modus Ponens

$$\frac{[A \leftarrow A;P] \Rightarrow B_1 \qquad \cdots \qquad [A \leftarrow A;P] \Rightarrow B_n}{[A \leftarrow B_1,\ldots,B_n;P] \Rightarrow A}\text{ MP}$$

Prefixing

$$\frac{[P] \Rightarrow A \qquad [A \leftarrow A;P;B_1] \Rightarrow B_1 \quad \ldots \quad [A \leftarrow A;P;B_n] \Rightarrow B_n}{[A \leftarrow B_1,\ldots,B_n;P] \Rightarrow A}\text{ PFX}$$

The axioms correspond to the fact that the empty goal immediately succeeds on any program. The soundness of the various rules of **S** can be seen as follows:

- The rule of *heterogeneous permutation* HP expresses the fact that, with respect to the relevant proof procedure, only the order of clauses with identical heads is relevant.
- To see the validity of the *modus ponens* rule MP, reason as follows. The premisses $[A \leftarrow A;P] \Rightarrow B_i$ express that the B_i can be successfully computed from $A \leftarrow A;P$, using general depth-first search. Also, the prefix $A \leftarrow A$ ensures that the B_i do not depend on A. (To see this, suppose that B_i does depend on A, that is, at some stage in the search for B_i from $A \leftarrow A;P$, the atom A occurs as a goal. At that point, in contradiction to the success of B_i from $A \leftarrow A;P$, the computation would get stuck in a loop because $A \leftarrow A$ is the 'uppermost' A-clause.) An evaluation of A in the context of $A \leftarrow B_1,\ldots,B_n;P$, will start with a (parallel) evaluation of the B_i. As the B_i do not depend on A, these evaluations in the context of $A \leftarrow B_1,\ldots,B_n;P$ amount to evaluations in the context of P — they will all succeed. Thus A will succeed from $A \leftarrow B_1,\ldots,B_n;P$.
- The soundness of the *prefixing rule* PFX depends on the following. First observe that, for any program Q and goal B, if the search for B from Q fails in finitely many steps, then the search for B from $Q;B \leftarrow$ will succeed. Also, if

the computation of B in the context of Q is successful, it will similarly succeed on $Q; B \leftarrow$. Conversely, suppose that the goal B succeeds on $Q; B \leftarrow$. Then either B already succeeds on a clause in Q with head B — that is, B succeeds on Q — or it successively fails on all the Q-clauses in Q, and finally succeeds via the fact clause $B \leftarrow$. To summarise, B succeeds on $Q; B \leftarrow$ iff B either succeeds or finitely fails on Q.
Thus the premisses $[A \leftarrow A; P; B_i] \Rightarrow B_i$ ensure that the search for any of the B_i, in the context of P, either succeeds or ends in finite failure, while the B_i do not encounter A as a subgoal during this search. An evaluation of A via the B_i in the context of P will thus either be successful (when all of the B_i succeed) or fail finitely (when some of the B_i fail finitely). In the latter case, the computation proceeds by an evaluation of A via the next A-clause. The assumption $[P] \Rightarrow A$ then ensures that the computation eventually succeeds. Thus the clause $A \leftarrow B_1, \ldots, B_n$ can be prefixed to the program P without destroying the success of a search for A.

3.1.1. EXAMPLE. A derivation of the sequent $[A \leftarrow B, C; A \leftarrow; B \leftarrow] \Rightarrow A$ in **S**:

$$\frac{\begin{array}[b]{c}[A \leftarrow A; B] \Rightarrow \top \\ [A; B] \Rightarrow A\end{array} \qquad \begin{array}[b]{c}[B \leftarrow B; A \leftarrow A; A; B] \Rightarrow \top \\ [B; A \leftarrow A; A; B] \Rightarrow B \\ [A \leftarrow A; B; A; B] \Rightarrow B \\ [A \leftarrow A; A; B; B] \Rightarrow B\end{array} \qquad \begin{array}[b]{c}[C \leftarrow C; A \leftarrow A; A; B] \Rightarrow \top \\ [C; A \leftarrow A; A; B] \Rightarrow C \\ [A \leftarrow A; C; A; B] \Rightarrow C \\ [A \leftarrow A; A; C; B] \Rightarrow C \\ [A \leftarrow A; A; B; C] \Rightarrow C\end{array}}{[A \leftarrow B, C; A; B] \Rightarrow A}$$

Here, the three subderivations all start with an axiom followed by an application of the modus ponens rule, and applications of the permutation rule HP. The last rule applied is PFX. □

Intuitively, the system **S** is also complete with respect to general depth-first search:
Suppose an atom A succeeds via the intended search mechanism from a program R. According to the inference mechanism, the 'uppermost' clause with head A, say $\mathcal{A}$, is tried first. The heterogenous permutation rule guarantees that we can 'reach' this clause. We distinguish two cases:

- Suppose A succeeds via $\mathcal{A}$. Then all the body atoms of $\mathcal{A}$ succeed in the context of R; moreover, none of these atoms depend on A (otherwise, the search for A would get stuck into a loop). Therefore, all the $\mathcal{A}$-body atoms must succeed in the context of the program in which the uppermost A-clause of R is substituted by the reflexive A-clause $A \leftarrow A$. This is, modulo applications of the heterogeneous permutation rule, precisely going from the conclusion of the modus ponens rule to its assumptions.
- Suppose now that A 'fails via' $\mathcal{A}$. Then, according to general depth-first search, it must succeed from the program R if the leftmost occurrence of the clause

$\mathcal{A}$ is deleted from it—this corresponds, modulo heterogeneous permutation, to the leftmost assumption of the prefixing rule. Also, failure via the 'uppermost' A-clause implies that none of the $\mathcal{A}$-body atoms goes into a loop — this corresponds to the rightmost assumptions of the prefixing rule.

We will formalize the above soundness and completeness arguments in Section 3.3. In order to do this, we will provide **S** with a formal semantics in the next section.

3.2 A Formal Computational Semantics

In order to formalise the above arguments for the soundness and completeness of **S**, we provide **S** with a formal semantics. The objects of this *computation semantics* are closely related to the appropriate search trees for general depth first search. Thus we will first have a look at these search trees.

Due to the parallellism in general depth-first search, the appropriate search trees for this computation mechanism are *and-or trees.* (In contrast with this, the appropriate search trees for Prolog computation are or-trees.) More specifically, and-or trees for general depth-first search have the following properties:

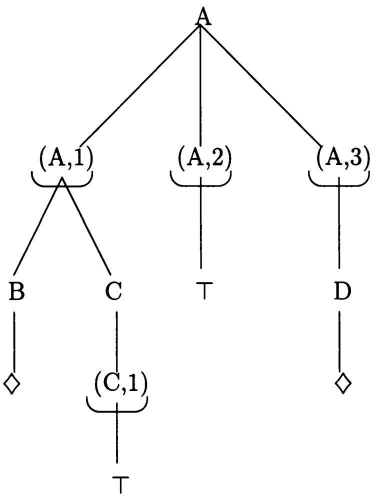
- the top node of the tree is an or-node, labelled with a proper atom, or a terminal node labelled $\top$;
- children of or-nodes are and-nodes, labelled with pairs of proper atoms and natural numbers, or terminal nodes $\Diamond$;
- children of and-nodes are or-nodes or terminal nodes $\top$;
- terminal nodes are labelled with $\top$ (corresponding to immediate success) or $\Diamond$ (corresponding to immediate failure).

We define the search trees for general depth first search as follows:

3.2.1. DEFINITION. An and-or tree T for general depth first search is the *search tree for a goal A from a program P via general depth first search* if it has the following additional properties:

- the top node of the tree is labelled with goal A;
- if an or-node has label B and there are $n > 0$ B-clauses in P, then it has n children (and-nodes), with respective labels $(B,1),\ldots,(B,n)$;
- if an or-node has label B different from $\top$ and there are no B-clauses in P, then it has as single child a terminal node labelled $\Diamond$;
- if an or-node has label $\top$, then it is terminal;
- if an and-node has label (B,m), then it has k children, where k is the number of body-atoms of the m-th leftmost B-clause in P, say $B \leftarrow D_1,\ldots,D_k$, and the labels of its children are $D_1,\ldots,D_k$, respectively. □

Observe that for every program P and every atom A there is exactly one associated general depth-first search and-or tree.
As an example, we give the and-or search tree for the atom A and the program $A \leftarrow B, C\ ;\ A \leftarrow\ ;\ A \leftarrow D\ ;\ C \leftarrow .$



The success or finite failure of a goal A from a program P is established by traversing the appropriate search-tree as determined by the computation procedure. For every goal A and program P there is a unique search-path which is traversed. Observe that, by the parallel nature of the selection rule under investigation, this path consists of parallel components. In order to determine success or finite failure of a goal, we can, instead of considering complete search trees, concentrate on the relevant part of the search-path needed to establish success or failure of a goal. If a goal succeeds or fails finitely, the appropriate search-path is finite, even though the search-tree itself need not be finite. As the success or finite failure of a goal is witnessed by the relevant search path, we can use these search paths as a formal semantics for the calculus **S**. For finite and completed search-paths we reserve the term *computation* — *successful computation* if it witnesses the success of a goal, *failing computation* if it witnesses finite failure. A computation will thus be the relevant part of a search tree. Moreover, a computation is (by definiton) finite. We will use the description of the computation mechanism to obtain an inductive definition of the set of all computations. Before we can give the definition of the computation semantics, we need an operator that, for each program P and each atom A, erases all those clauses from P whose head is different from A.

3.2.2. DEFINITION. For atoms A , clauses $\mathcal{C}$, and programs P and R:

$$\begin{aligned} \emptyset_A &= \emptyset \\ \mathcal{C}_A &= \begin{cases} \mathcal{C} & \text{if } \mathcal{C} \text{ has head } A, \\ \emptyset & \text{otherwise} \end{cases} \\ (P;R)_A &= P_A; R_A. \end{aligned}$$

□

We are now in a position to give the inductive definition of the notion of formal computation.

3.2.3. DEFINITION.

1. $\square$ is a *successful computation* for $\top$ from P .
2. $A[\Diamond]$ is a *failing computation* for A from P if $P_A = \emptyset$
3. Suppose that

$$P_A = A \leftarrow B_1^1, \ldots, B_{f(1)}^1; \ \ldots \ ; A \leftarrow B_1^n, \ldots, B_{f(n)}^n; R,$$

where R is a program (in which only clauses with head A occur), and let, for $1 \leq i \leq n$ and $1 \leq j \leq f(i)$, Π_j^i be the (successful or failing) computations for B_j^i from P.
Then

$$A \left[\Pi_1^1 | \cdots | \Pi_{f(1)}^1\right] \cdots \left[\Pi_1^n | \cdots | \Pi_{f(n)}^n\right]$$

is

- a successful computation for A from P if

$$\begin{array}{rl} & \forall i < n \ \exists j \leq f(i) \text{ such that } \Pi_j^i \text{ is a failing computation,} \\ \text{while} & \forall j \leq f(n) \ \Pi_j^n \text{ is a successful computation.} \end{array}$$

- a failing computation for A from P if

$$\begin{array}{rl} & R = \emptyset \\ \textit{and} & \forall i \leq n \ \exists j \leq f(i) \ \Pi_j^i \text{ is a failing computation.} \end{array}$$

4. Π is a *computation* iff it is a successful computation or a failing computation.
$\square$

3.2.4. EXAMPLE.

1. Consider the program $P = A \leftarrow B, C; A \leftarrow; B \leftarrow; A \leftarrow A$. The (successful) computation for A from P will be $A\,[B\,[\square]\,|C\,[\Diamond]]\,[\square]$.
2. Consider the program $R = A \leftarrow A; A \leftarrow B, C; A \leftarrow; B \leftarrow$. There is *no* computation for A from R. $\square$

The close connection between computations and the inference mechanism above is expressed in the following lemma.

3.2.5. LEMMA. *There exists a successful computation for A from P iff a general depth-first search for A from the program P succeeds. Existence of a failing computation likewise corresponds to the finite failure of general depth-first search.*

The remainder of the present section is devoted to the introduction of some notions related to formal computations and to a short discussion of some relevant properties of formal computations.

3.2.6. DEFINITION. (Subcomputations)

1. Every computation is a subcomputation of itself.

2. If
$$\Pi = A\left[\Pi^1_1 |\cdots| \Pi^1_{f(1)}\right] \cdots \left[\Pi^n_1 |\cdots| \Pi^n_{f(n)}\right]$$
is a computation, then all subcomputations of Π^i_j, for $1 \leq i \leq n, 1 \leq j \leq f(i)$, are subcomputations of Π .
3. Σ is a proper subcomputation of Π if it is a subcomputation of Π and not equal to Π. □

3.2.7. Definition. (Depth of a computation)

1. $\mathtt{depth}(\Box) = \mathtt{depth}(A[\Diamond]) := 1$;
2. In all other cases, $\mathtt{depth}(\Pi)$ is 1 + the maximum of the depths of the proper subcomputations of Π. □

3.2.8. Definition. For programs P and atoms A, P^{-A} is the program obtained by deleting from P the leftmost clause with head A. Iterations of the deletion operator are defined as follows:

$$\begin{aligned} P^{0,A} &:= P \\ P^{1,A} &:= P^{-A} \\ P^{n+1,A} &:= (P^{n,A})^{-A} \end{aligned}$$
□

We are now able to state some relevant properties of computations. We omit the proofs.

- **Uniqueness**

  There exists at most one computation for each program P and each atom A. A failing computation is not a successful computation and vice versa.
- **Finiteness**

  Each computation is a finite object.
- **Occurrence**

  If Π is a computation for A from P, then A does not occur in any of the proper subcomputations of Π.
- **Transformation**
  1. If Π is a successful computation for A from P, then Π is a successful computation for A from $P; A \leftarrow$.
  2. If Π is a failing computation for A from P, then $\Pi[\Box]$ is a successful computation for A from $P; A \leftarrow$.
  3. If Π is a (successful) computation for A from $P; A \leftarrow$, then
     (a) Π is a successful computation for A from P, or
     (b) $\Pi = \Sigma[\Box]$ and Σ is a failing computation for A from P.
- **Inessential Difference**

  If Π is a computation for A from P, and R is a program such that, for every atom B occurring in Π, $P_B = R_B$, then Π is also a computation for A from R.

3.3 Soundness and Completeness

In the present section, we set out to prove the correctness of the calculus **S** for the general depth first search procedure. We will first prove soundness and completeness of **S** with respect to the computation semantics (Lemmas 3.3.1 and 3.3.2). The correctness of **S** with respect to general depth first search is an immediate consequence of these results and the correctness of the computation semantics with respect to the search procedure (Lemma 3.2.5).

3.3.1. LEMMA. (Soundness) *If* $\mathbf{S} \vdash [P] \Rightarrow A$ *then there exists a successful computation for* A *from* P.

PROOF. (By induction on the depth of derivations in **S**.)

- AXIOMS $\square$ is a successful computation for $\top$ from P, while $[P] \Rightarrow \top$ is the only possible derivation of depth 1.
- HETEROGENEOUS PERMUTATION The notion of successful computation is by definition insensitive to heterogeneous permutation in programs. Therefore this rule is trivially sound.
- MODUS PONENS Suppose that the sequents $[A \leftarrow A; P] \Rightarrow B_i$ are derivable in **S**, for $1 \leq i \leq n$. By inductive hypothesis there are successful computations Π_i for B_i from $A \leftarrow A; P$. Observe that, by the finiteness property, the atom A does not occur in any of the Π_i. Thus, by the inessential difference property, the Π_i are successful computations for B_i from P. By definition of successful computation, $A[\Pi_1|\cdots|\Pi_n]$ is a successful computation for A from the program $A \leftarrow B_1, \ldots, B_n; P$.
- PREFIXING Suppose that $[P] \Rightarrow A$ and $[A \leftarrow A; P; B_i \leftarrow] \Rightarrow B_i$ (for $1 \leq i \leq n$) are derivable in **S**. By inductive hypothesis there are $\sigma_1, \ldots, \sigma_m$ such that $A[\sigma_1]\cdots[\sigma_m]$ is a successful computation for A from P. Also, there are successful computations Π_i for B_i from $A \leftarrow A; P; B_i \leftarrow$. By the finiteness property, the inessential difference property and transformation property (3), the Π_i can be transformed into (failing or successful) computations Π_i^* for B_i from $A \leftarrow B_1, \ldots, B_n; P$. If all the Π_i^* are successful computations, then (by definition) $A[\Pi_1^*|\cdots|\Pi_n^*]$ is a successful computation for A from $A \leftarrow B_1, \ldots, B_n; P$. Otherwise, again by definition, $A[\Pi_1^*|\cdots|\Pi_n^*][\sigma_1]\cdots[\sigma_m]$ is a successful computation for A from $A \leftarrow B_1, \ldots, B_n; P$. $\square$

3.3.2. LEMMA. (Completeness) *If* Π *is a successful computation for* A *from* P, *then* $\mathbf{S} \vdash [P] \Rightarrow A$.

PROOF. (By induction on the depth of computations.)
There is only one successful computation of depth 1, $\square$, which corresponds to derivability of $[P] \Rightarrow \top$.
Suppose the claim holds for every successful computation of depth $< k$.
Let

$$\Pi = A\left[\Pi_1^1|\cdots|\Pi_{f(1)}^1\right]\cdots\left[\Pi_1^n|\cdots|\Pi_{f(n)}^n\right]$$

be a successful computation for A from P, such that $\mathbf{depth}(\Pi) = k$.
We distinguish two cases: A succeeds via the leftmost A-clause of P, that is, $n = 1$, or A succeeds via some further A-clause in P, that is, $n > 1$.

- $n = 1$, that is, $\Pi = A\left[\Pi_1 \mid \cdots \mid \Pi_{f(1)}\right]$.
  By definition, the Π_i are successful computations, say for B_i from P. Also, $P_A = A \leftarrow B_1, \ldots, B_n; R$ for some R.
  By the occurrence property for A and the inessential difference property, the Π_i are successful computations for B_i from $A \leftarrow A; P^{-A}$. By inductive hypothesis, we can conclude that, for all i, $\left[A \leftarrow A; P^{-A}\right] \Rightarrow B_i$ is derivable. An application of Modus Ponens yields a derivation of $\left[A \leftarrow B_1, \ldots, B_n; P^{-A}\right] \Rightarrow A$. An extension, if necessary, with applications of the heterogeneous permutation rule then results in a derivation of $[P] \Rightarrow A$.
- $n > 1$.
  We will show by induction on t that $\left[P^{t,A}\right] \Rightarrow A$ is derivable for all $t < n$. First observe that, by definition of successful computation and the inessential difference properties, $A[\Pi_1^n \mid \cdots \mid \Pi_{f(n)}^n]$ is a successful computation for A from $P^{n-1,A}$ — in particular, A succeeds via the leftmost A-clause of $P^{n-1,A}$. By the above case, this implies that $\left[P^{n-1,A}\right] \Rightarrow A$ is derivable.
  Now suppose that for some $s \leq n-1$, $\left[P^{s,A}\right] \Rightarrow A$ is derivable. Now, by definition, the Π_i^s are (successful or failing) computations for, say B_i^s from P, for $1 \leq i \leq f(s)$. By the inessential difference property, they are also computations for B_i^s from $A \leftarrow A; P^{s,A}$. By the transformation property, these can be transformed into successful computations Σ_i^s for B_i^s from $A \leftarrow A; P^{s,A}; B_i^s$. By inductive hypothesis, it follows that the sequents $\left[A \leftarrow A; P^{s,A}; B_i^s\right] \Rightarrow B_i^s$ are derivable. Apply PFX to the derivable sequents $\left[A \leftarrow A; P^{s,A}; B_i^s\right] \Rightarrow B_i^s$ and $\left[P^{s,A}\right] \Rightarrow A$, and derivability of $\left[P^{s-1,A}\right] \Rightarrow A$ follows (possibly after some applications of Heterogeneous Permutation).
  This concludes the proof of the theorem. □

The correctness of **S** for the general depth first search is an immediate consequence of the above soundness and completeness results and the correctness of the computation semantics with respect to the search procedure (Lemma 3.2.5).

3.3.3. THEOREM.
S $\vdash [P] \Rightarrow A$ *iff* A *succeeds on* P *via general depth first search.*

3.4 Decidability

Decidability of **S** could be easily inferred from properties of computations and the soundness and completeness result for **S**. Instead, we choose to prove decidability using properties of derivations in **S**. The advantage of taking this route is that it gives us some more insight in derivations in **S**. The section on normalisation is a proof-theoretic preliminary for the decidability proof.

3.4.1 Normalisation

A sequent derivable in **S** can have several derivations. While the rule for heterogeneous permutation can cause obvious variation, more interesting are variations due to applications of the prefixing rule. As an example, consider the derivable sequent $[A \leftarrow C; A \leftarrow B; C \leftarrow; B \leftarrow] \Rightarrow A$. The following are (the relevant parts of) two of its possible derivations. The non-exhibited parts of the derivations consist of axioms followed by one application of MP, and applications of HP.

(1)
$$\frac{\begin{array}{c}\vdots\\ [A \leftarrow A; A \leftarrow B; C; B] \Rightarrow C\end{array}}{[A \leftarrow C; A \leftarrow B; C; B] \Rightarrow A}\,\text{MP}$$

(2)
$$\frac{\dfrac{\begin{array}{c}\vdots\\ [A \leftarrow A; C; B] \Rightarrow B\end{array}}{[A \leftarrow B; C; B] \Rightarrow A}\,\text{MP} \quad \begin{array}{c}\vdots\\ [A \leftarrow A; A \leftarrow B; C; B; C] \Rightarrow C\end{array}}{[A \leftarrow C; A \leftarrow B; C; B] \Rightarrow A}\,\text{PFX}$$

The difference between these two different derivations can be thought of as follows. In the uppermost derivation (1), the clause $A \leftarrow C$ is prefixed to the program $A \leftarrow B$; $C \leftarrow$; $B \leftarrow$ by an application of Modus Ponens. In derivation (2), the same clause is prefixed by an application of PFX, which is in its turn preceded by the prefixing of the clause $A \leftarrow B$ using Modus Ponens. As witnessed by derivation (1), the application of the prefixing rule can be considered unnecessary, as it can be 'replaced' by an application of Modus Ponens. This example generalises to a notion of *normal derivations*, that is, derivations without unnecessary applications of PFX. There is a direct correspondence between this notion of normality for derivations and the computational semantics: a computation is a direct translation of a normal derivation and vice versa. The proof of the completeness theorem in fact gives us an algorithm to construct normal derivations from a given successful computation. Thus, as an immediate consequence of the soundness theorem, the proof of the completeness theorem, and the uniqueness of successful computations, we have the following result:

3.4.1. PROPOSITION. *Every* **S***-derivable sequent has a unique normal derivation (up to applications of HP) and there is an algorithm that transforms every derivation into a normal derivation, which is unique up to applications of HP.*

3.4.2 Decidability

We will prove decidability of **S** via an analysis of possible derivations.

First of all, we can in fact restrict applications of the rules MP and PFX, by adding a side condition to these rules. This follows from the following proposition.

3.4.2. PROPOSITION. *If* $[P] \Rightarrow A$ *is derivable in* **S***, then* A *does not occur in the body of the leftmost* A*-clause of* P*.*

PROOF. By induction on the depth of derivations. Let δ be a derivation of depth 1. Then δ is an axiom $[P] \Rightarrow \top$. By definition, $\top$ does not occur as the head of clauses in P. So the consequent trivially holds in this case. Let δ be a derivation of depth n, and suppose that the proposition is true for all derivations of depth $< n$.
Suppose the last rule applied in δ is HP,

$$\frac{[P; \mathcal{C}; \mathcal{B}; R] \Rightarrow A}{[P; \mathcal{B}; \mathcal{C}; R] \Rightarrow A}\ \text{HP}$$

By the inductive hypothesis, A does not occur in the body of the leftmost A-clause $\mathcal{A}_1$ of P; $\mathcal{C}$; $\mathcal{B}$; R. But $\mathcal{B}$ and $\mathcal{C}$ have different heads. Consequently, the leftmost A-clause of P; $\mathcal{B}$; $\mathcal{C}$; R is also $\mathcal{A}_1$.
Suppose that the last rule applied in δ is Modus Ponens,

$$\frac{[A \leftarrow A; P] \Rightarrow B_1 \qquad \cdots \qquad [A \leftarrow A; P] \Rightarrow B_n}{[A \leftarrow B_1, \ldots, B_n; P] \Rightarrow A}\ \text{MP}$$

Suppose, by contraposition, that $A = B_i$ for some i. Then the i-th assumption of this instance of Modus Ponens reads:

$$[A \leftarrow A;\ P] \Rightarrow A.$$

This contradicts the inductive hypothesis. The same argument applies if the last rule applied is PFX. □

The above proposition shows that in derivations in **S**, PFX and MP are only applied if $A \neq B_i$, for all $i \in [1, n]$. Consequently, we can consider the following modification of **S**. Add, to the rules MP and PFX, the side condition:

$$A \neq B_i, \qquad \text{for all } i \in [1, n]$$

Let now **S**' be the calculus consisting of the axioms of **S** and the rule for Heterogenous Permutation, and the rules MP and PFX with the above side condition.

Then, by the remarks above, the modified calculus **S**' is equivalent to **S** in the sense that a sequent is derivable in **S** iff it is derivable in **S**'. That is, the following holds:

3.4.3. PROPOSITION. *δ is a derivation in* **S** *iff δ is a derivation in* **S**'.

Thus, we can restrict our considerations to **S**'.

Second, in **S**'-derivations of a sequent $[P] \Rightarrow A$, the number of applications of MP and PFX in each branch is restricted by the number of non-reflective clauses in P. This in fact follows from the restrictions on MP and PFX is **S**'. Observe that the modified MP and PFX rules of **S**' cannot prefix reflexive clauses $A \leftarrow A$.

Define the *weight* $v(\mathcal{A})$ of a clause $\mathcal{A}$ as follows:

$v(\mathcal{A}) = 0$ iff $\mathcal{A}$ is a reflexive clause $A \leftarrow A$,

$v(\mathcal{A}) = 1$ otherwise.

Define the weight $v(P)$ of a program P as the sum of the weights of the clauses in P. Define the weight $v([P] \Rightarrow A)$ of a sequent $[P] \Rightarrow A$ as the weight of P.

For any instance of the MP rule in **S**', the weight of the assumptions is strictly smaller than the weight of the conclusion. (In contrast, this does not hold for **S**.) Also, the weight of the left assumption of any instance of the PFX rule in **S**' is strictly smaller than the weight of the conclusion. In contrast, the weight of the right assumptions of the **S**'-prefixing rule is equal to the weight of the conclusion. However, going from conclusion to right assumptions in PFX, a clause $A \leftarrow B_1, \ldots, B_n$ is traded for a fact-clause $B \leftarrow$, and the latter can in its turn only have been introduced by an application of MP. As a result of these observations, we have the following.

3.4.4. PROPOSITION. *Let δ be an* **S**'*-derivation of the sequent $[P] \Rightarrow A$, and let d be a branch in δ. Then the total number of applications of MP and PFX in d is smaller than or equal to $2 \cdot v([P] \Rightarrow A)$.*

Third, we can restrict the depth of derivations, by restricting the number of applications of the HP rule. We use the following proposition, whose proof is easy.

3.4.5. PROPOSITION. *Let P be a program $P = C_1, \ldots C_n$, and let k be minimal such that C_k is an A-clause.*
Then $[P] \Rightarrow A$ is derivable iff the sequent $[C_k, C_1, \ldots, C_{k-1}, C_{k+1}, \ldots, C_n] \Rightarrow A$
(a) is an instance of an axiom, or
(b) has a derivation ending in an application of MP or PFX.
Consequently, $[P] \Rightarrow A$ is derivable iff it has a derivation ending in at most k consecutive applications of HP. Trivially, $k \leq n$.

Now define the *sequent-length* $\#([P] \Rightarrow A)$ as the total number of clauses in P. That is, if P is the program $C_1, \ldots, C_n$, then $\#([P] \Rightarrow A) = n$. Observe that the number of clauses of the antecedents decreases by at most one in all **S**-rules — more specifically, an application of the HP rule does not decrease the sequent-length, while by applications of MP and PFX the sequent-length decreases with at most 1.

By the Propositions 3.4.4 and 3.4.5, we can conclude that a sequent $[P] \Rightarrow A$ is derivable in **S** iff there is a derivation (in **S**') of depth smaller than $2n^2$, where $n = \#([P] \Rightarrow A)$.

Fourth, the branching degree of possible derivations of a sequent is limited. In particular, let m be the maximal length of the bodies of clauses in P. That is, for every clause $C \leftarrow B_1, \ldots, B_k$ in P, $k \leq m$. (Observe that, in the course of a derivation, this number increases.) Then the branching degree of a derivation of $[P] \Rightarrow A$ is at most $m + 1$ (due to applications of PFX).

In addition, suppose that s is a sequent occurring in a derivation of $[P] \Rightarrow A$. Then the sequent-length of s is at most $3n$, where $n = v([P] \Rightarrow A)$. Also, the clauses occurring in s either occur in P, or are of the form $C \leftarrow C$, where C occurs as the head of a clause in P, or are of the form $C \leftarrow$, where C occurs in the body of a clause of P.

We can conclude that

3.4.6. THEOREM. **S** *is decidable.*

An open question is the structural complexity of **S**.

3.5 A Substructural Landscape

In the present and the subsequent section, we will investigate the substructural nature of **S**. In particular, we will study the effect of extending the calculus **S** with various structural rules and cut rules.

In general, we can distinguish two different outcomes of extending a calculus C with a rule R. First, the extension C + R can be strictly stronger than the original calculus C, that is, in C + R more sequents are derivable than in C. Second, the extension with R can be equivalent to C with respect to derivability. In the latter case, we call R *sound* with respect to C and C + R *conservative* over C; that is, for all sequents s, $C + R \vdash s$ iff $C \vdash s$.

In Figure 3.1, we list the proper formulations of the classical structural rules (Exchange EX, Contraction, and Weakening W) and the usual cut rule (Classical Cut CC) in the context of **S**. As the antecedents of the sequents in **S** are lists rather than multisets, the proper formulation of the usual contraction rule has to be modified in the context of **S**. More precisely, in an order sensitive calculus like **S**, the rule for contraction splits in two distinct versions Rightward Contraction RC and Leftward Contraction LC. We will show that extending **S** with any of these classical rules, with the exception of Rightward Contraction, results in a strictly stronger calculus (Section 3.5.2 and 3.6).

In Section 3.5.1 and 3.6 we will discuss several weak versions of these classical rules, listed in Figure 3.2. Rightward Extension RE and Atomic Monotonicity AM are both special cases of the full Weakening rule W. Full Context Cut FCC and Partial Context Cut PCC are special cases of Classical Cut CC. For any of the weaker structural rules listed in Figure 3.2, we will show that they are sound with respect to **S**.

In our discussion of sound rules, we will further distinguish admissible and derivable rules (cf. [Dos92]).

3.5.1. DEFINITION.
A rule R is *admissible* in a calculus C if the conclusion of R is derivable from C when the assumptions of R are derivable from C.
A rule R is *derivable* in a calculus C if the conclusion of R is derivable from its assumptions, using axioms and rules of C.

The relevance of derivability and admissibility is expressed in the following proposition:

3.5.2. PROPOSITION. *Let C be a calculus and R a rule. If R is derivable or admissible in C, then $C + R$ is a conservative extension of C. Conversely, if $C + R$ is a conservative extension of C, then R is admissible in C.*

The following proposition states some easy properties of these two concepts.

Exchange

$$\frac{[P;\mathcal{B};\mathcal{C};R] \Rightarrow A}{[P;\mathcal{C};\mathcal{B};R] \Rightarrow A}\,\text{EX}$$

Weakening

$$\frac{[P;R] \Rightarrow A}{[P;\mathcal{B};R] \Rightarrow A}\,\text{W}$$

Contraction

$$\frac{[P;\mathcal{B};Q;\mathcal{B};R] \Rightarrow A}{[P;Q;\mathcal{B};R] \Rightarrow A}\,\text{LC} \qquad \frac{[P;\mathcal{B};Q;\mathcal{B};R] \Rightarrow A}{[P;\mathcal{B};Q;R] \Rightarrow A}\,\text{RC}$$

Classical Cut

$$\frac{[Q] \Rightarrow C \qquad [P;C \leftarrow;R] \Rightarrow A}{[P;Q;R] \Rightarrow A}\,\text{CC}$$

Figure 3.1: Classical rules

Rightward Extension

$$\frac{[P;\mathcal{B};Q;R] \Rightarrow A}{[P;\mathcal{B};Q;\mathcal{B};R] \Rightarrow A}\,\text{RE}$$

Atomic Monotonicity

$$\frac{[P;R] \Rightarrow A}{[P;B \leftarrow;R] \Rightarrow A}\,\text{AM}$$

Full Context Cut

$$\frac{[P;R] \Rightarrow C \qquad [P;C \leftarrow;R] \Rightarrow A}{[P;R] \Rightarrow A}\,\text{FCC}$$

Partial Context Cut

$$\frac{[P] \Rightarrow C \qquad [Q;C \leftarrow;R;Z] \Rightarrow A}{[Q;R;Z] \Rightarrow A}\,\text{PCC} \qquad \text{where } P \subseteq Q;R \text{ and } R_C = \emptyset.$$

Figure 3.2: Substructural rules

3.5.3. PROPOSITION. *Let C be a calculus and R a derivation rule. The following hold:*
(1) If R is derivable in C, then it is admissible in C.
(2) If R is derivable in C, then it is derivable in every rule extension of C.
(3) If R is derivable in C, then it is admissible in any rule extension of C.

In contrast, admissibility does not imply derivabilility. In Section 3.5.1 we will see some examples of rules that are admissible for **S**, but not derivable.

We also need the following related concepts, to compare the relative strength of calculi:

3.5.4. DEFINITION. Let S and T be two Gentzen calculi.

1. $S \preceq T$ iff $X \vdash s \Rightarrow T \vdash s$, for all sequents s.
2. $S \prec T$ iff $S \preceq T$ and there is a sequent s such that $T \vdash s$, while $S \not\vdash s$.
3. $S \equiv T$ iff $S \preceq T$ and $T \preceq S$.

3.5.1 Admissible rules for S

In [vB92], a calculus for general depth first search was discussed wich consisted of (a minor variant of) the rules of **S** plus the rules Rightward Contraction and Rightward Extension. Intuitive soundness and completeness proofs were given there. Using the computation semantics we show that these rules are indeed admissible in **S** (Proposition 3.5.6 and 3.5.7). That raises the question whether these rule are also derivable in **S**. By giving rule-extensions of **S** in which they are not admissible, we show that this is not the case.

We first extablish the following useful lemma.

3.5.5. LEMMA. *Let Π be the successful (failing) computation for B from the program $P = Q; \mathcal{B}; R; \mathcal{B}; Z$, where $\mathcal{B}$ is a clause with head B. Then there is a successful (failing) computation Π' for B from $Q; \mathcal{B}; R; Z$.*

PROOF. Let Π satisfy the conditions of the lemma, and let $\Pi = B[\delta_1]\cdots[\delta_n]$. Also, let the leftmost and rightmost exhibited $\mathcal{B}$ in the formulation of the lemma be the m-th and k-th B-clause in Π, respectively.

We distinguish the following cases:

1. $n < k$. Then Π is also the (successful) computation for B from $Q; \mathcal{B}; R; Z$. So take $\Pi' = \Pi$.
2. $n > k$. Then, by definition of computations, δ_k consists of a list of parallel computations, among which at least one is failed. So consider $\Pi' = B[\delta_1]\cdots[\delta_{k-1}][\delta_{k+1}]\cdots[\delta_n]$. Π' is the computation for B from $Q; \mathcal{B}; R; Z$, and Π' is successful iff Π is successful.
3. $n = k$. Observe that $\delta_m = \delta_k$. Therefore, Π is a failing computation, and $\Pi' = B[\delta_1]\cdots[\delta_{k-1}]$ is the failing computation for B from $Q; \mathcal{B}; R; Z$. □

3.5.6. PROPOSITION. *Rightward Contraction is admissible in* **S**.

PROOF. Let Σ be a successful computation for A from $Q; \mathcal{B}; R; \mathcal{B}; Z$, where $\mathcal{B}$ is a clause with head B. Let Π and Π' be the computations for B from respectively $Q; \mathcal{B}; R; \mathcal{B}; Z$ and $Q; \mathcal{B}; R; Z$. By the above Lemma 3.5.5, Π is successful iff Π' is successful.
We distinguish the following two cases:
B does not occur in Σ. Then, by the inessential difference property, Σ is also the successful computation for A from $Q; \mathcal{B}; R; Z$.
B does occur in Σ. Replacing all occurrences of Π in Σ by Π' results in a successful computation for A from $Q; \mathcal{B}; R; Z$.
The proposition now follows from the definition of computations and by soundness and completeness of **S** with respect to computations. □

By similar arguments, we have the following.

3.5.7. PROPOSITION. *Rightward Extension (RE) is admissible in* **S**.

In contrast, the following holds.

3.5.8. PROPOSITION. *Neither Rightward Extension nor Rightward Contraction are derivable rules in* **S**.

PROOF. By Proposition 3.5.3(3), it suffices to find rule extensions of **S** in which Rightward Extension RE and Rightward Contraction RC are not admissible. In particular, by Proposition 3.5.2 it suffices to find rules R_E and R_C and sequents T_E and T_C such that

1. $\mathbf{S} + \mathrm{R}_E + \mathrm{RE} \vdash T_E$, while $\mathbf{S} + \mathrm{R}_E \not\vdash T_E$;
2. $\mathbf{S} + \mathrm{R}_C + \mathrm{RC} \vdash T_C$, while $\mathbf{S} + \mathrm{R}_C \not\vdash T_C$.

Ad 1). Take for R_E the following rule:

$$\frac{[P] \Rightarrow A}{[A \leftarrow A; P] \Rightarrow A}\ \mathrm{R}_E \qquad \text{if } P_A = A \leftarrow,$$

and take for T_E the sequent $[A \leftarrow A; A \leftarrow; A \leftarrow] \Rightarrow A$. The derivation of T_E in $\mathbf{S}+$ R_E + RE is as follows:

$$\cfrac{\cfrac{\cfrac{[A \leftarrow A] \Rightarrow \top}{[A \leftarrow] \Rightarrow A}\ \mathrm{MP}}{[A \leftarrow A; A \leftarrow] \Rightarrow A}\ \mathrm{R}_E}{[A \leftarrow A; A \leftarrow; A \leftarrow] \Rightarrow A}\ \mathrm{RE}$$

To show that $\mathbf{S} + \mathrm{R}_E \not\vdash T_E$, suppose that δ is a derivation of T_E in $\mathbf{S}$ + RE. T_E is not the conclusion of an application of R_E or HP. Thus it must be the conclusion of either MP or PFX. In the first case, the single assumption of this rule is T_E itself, in the latter case, one of the assumptions is the sequent $[A \leftarrow A;\ A \leftarrow;\ A \leftarrow;\ A \leftarrow] \Rightarrow A$. Either of these sequents can again only be the conclusion of MP or PFX. Thus δ must contain an infinite branch.
This concludes the proof of 1).

Ad 2). Take for R_C the following rule:

$$\frac{[P] \Rightarrow A}{[A \leftarrow A; P] \Rightarrow A}\,R_C \qquad \text{if for some } Y,\ P_A = Y; A \leftarrow A; A \leftarrow,$$

and let T_C be the sequent $[A \leftarrow A;\ A \leftarrow] \Rightarrow A$. Now T_C and R_C are as in (2), providing a counterexample to derivability of Rightward Extension: We have the following derivation of T_C in **S**+ R_C + RC:

$$\cfrac{\cfrac{\cfrac{\cfrac{[A \leftarrow A; A \leftarrow A; A \leftarrow] \Rightarrow \top}{[A \leftarrow; A \leftarrow A; A \leftarrow] \Rightarrow A}\,\text{MP}}{[A \leftarrow A; A \leftarrow; A \leftarrow A; A \leftarrow] \Rightarrow A}\,R_C}{[A \leftarrow A; A \leftarrow; A \leftarrow] \Rightarrow A}\,\text{RC}}{[A \leftarrow A; A \leftarrow] \Rightarrow A}\,\text{RC}$$

The arguments used above in case 1) also apply here, to show that **S**+ $R_C \nvdash T_C$. This concludes the proof of 2). □

Another sound but non-derivable rule is Atomic Monotonicity, which is obtained by restricting Weakening to fact-clauses, see Figure 2. The intuitive argument for the soundness of Atomic Monotonicity is the following: The addition of a clause to a program only destroys success if the added clause introduces a loop; a fact-clause does not introduce any loops. Again the rule R_E and the clause T_E defined above serve as a counterexample to derivability of Atomic Monotonicity.

3.5.2 Exchange, Contraction, Weakening

Let us turn to the classical structural rules Exchange, Contraction (Leftward and Rightward), and Weakening. Extension of **S** with any of these rules results in a strictly stronger system:

3.5.9. PROPOSITION.

$$\begin{array}{lcl} \mathbf{S} & \prec & \mathbf{S}+\text{EX} \\ \mathbf{S} & \prec & \mathbf{S}+\text{LC}+\text{RC} \\ \mathbf{S} & \prec & \mathbf{S}+\text{W} \end{array}$$

PROOF. Consider the sequent $S = [A \leftarrow A; A \leftarrow] \Rightarrow A$. S is derivable in any of these three extensions of **S**, but not in **S**. □

The addition of Exchange to **S** corresponds to a shift from depth first search to breadth first search. That is, **S** + EX corresponds to classical logic (CL) for Horn clause theories. While it is not clear at first sight to what search procedure addition of Contraction or Weakening corresponds, it turns out that Exchange, Contraction, and Weakening are equivalent over **S**:

3.5.10. PROPOSITION. **S** + EX ≡ **S** + LC + RC ≡ **S** + W.

PROOF. We will show that any of the three calculi **S** + EX, **S** + W, and **S** + LC + RC, corresponds to breadth-first search in and-or search-trees. More precisely, we will show that a sequent $[P] \Rightarrow A$ is derivable in any the above calculi iff there exists a successbranch in the and-or search tree for A from P. We will use the following notion of formal breadth-first (bf) computations, which formalises the notion of success-branch in and-or trees:

- $\Box$ is a bf-computation for $\top$ from P.
- $A[\Pi_1|\cdots|\Pi_n]$ is a bf-computation for A from P if
    1. there is a clause $A \leftarrow B_1, \ldots, B_n$ in P such that, for $i \in [1,n]$, Π_i is a bf-computation for B_i from P;
    2. A does not occur in any of the Π_i;
    3. If Σ and Γ are bf-computations for some C from P, such that both Σ and Γ are subcomputations (using the obvious notion of subcomputation) of Π, then $\Sigma = \Gamma$.

Clearly, there is a (not necessarily unique) bf-computation for A from P iff breadth-first search with parallel goal processing for A from P succeeds. By proving soundness and completeness of each of the above calculi with respect to the breadt-first computation semantics, we show their mutual equivalence. Breadth-first computations have the following properties, which will be useful for proving soundness and completeness:
Let Π be a bf-computation for A from P. Then there is a *minimal* program Q such that

1. $Q \subseteq P$;
2. Every atom $\neq \top$ that occurs in Q, occurs exactly once as the head of a clause in Q;
3. Π is a bf-computation for every program $R \supseteq Q$.
4. Q is unique modulo permutation of clauses.

Soundness
We show that each of the rules of S and W, EX, LC, and RC, are sound with respect to bf-computations. Soundness of the axioms is immediate. Soundness of HP and EX follows from the fact that the bf-computations are insensitive to the relative order of clauses in a program. Soundness of PFX, W, LC, and RC follows easily from the above properties of bf-computations. Consider MP,

$$\frac{[A \leftarrow A; P] \Rightarrow B_1 \qquad \cdots \qquad [A \leftarrow A; P] \Rightarrow B_n}{[A \leftarrow B_1, \ldots, B_n; P] \Rightarrow A}\,\mathrm{MP}$$

Suppose that for $i \in [1,n]$, Π_i is a bf-computation for B_i from $A \leftarrow A; P$.
By the above properties of bf-computations, Π_i is a bf-computation for B_i from $A \leftarrow B_1, \ldots, B_n; P$, for $i \in [1,n]$. Without loss of generality we can assume that, if Σ_i and Σ_j are subcomputations of Π_i and Π_j, respectively, for an atom C, then Σ_i is identical to Σ_j. Therefore, if A does not occur in any of the Π_i, $A[\Pi_1|\ldots|\Pi_n]$ is a bf-computation for A from $A \leftarrow B_1, \ldots, B_n; P$. Suppose A occurs in Π_i. That is, there

is a subcomputation Σ of Π, which is a bf-computation for A from P. By the above properties of bf-computations, Σ is a bf-computation for A from $A \leftarrow B_1, \ldots, B_n; P$.

Completeness

By induction on the (natural) depth of bf-computations. Suppose that for all atoms A, programs P, if there is a bf-computation for A from P of depth $< n$, then the sequent $[P] \Rightarrow A$ is derivable from any of the relevant calculi. Let $\Pi = A[\Pi_1|\cdots|\Pi_n]$ be a bf-computation of depth n for A from P. Let $Q \subseteq P$ be the minimal program associated with Π, and let Q_A be $A \leftarrow B_1, \ldots, B_n$. Let Q' be the result of deleting Q_A from Q, and let P' be the result of deleting the leftmost occurrence of Q_A from P. Then, by the above properties of bf-computations, we find that, for $i \in [1, n]$, Π_i is a bf-computation for B_i from any of

1. $A \leftarrow A; Q'$,
2. $A \leftarrow A; P'$, and
3. $A \leftarrow A; P$.

Therefore, by the inductive hypothesis, we find corresponding derivations v_i, ϵ_i, and σ_i, in respectively **S** + W, **S** + EX, and **S** + LC + RC. Indicating consecutive applications of a rule R by R*, we find the following derivations of the sequent $[P] \Rightarrow A$, respecively in **S** + W, **S** + EX, and **S** + LC + RC.

$$\dfrac{\dfrac{\dfrac{\overset{v_1}{[A \leftarrow A; Q'] \Rightarrow B_1} \quad \cdots \quad \overset{v_n}{[A \leftarrow A; Q'] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; Q'] \Rightarrow A}\,\text{MP}}{[Q] \Rightarrow A}\,\text{HP*}}{[P] \Rightarrow A}\,\text{W*}$$

$$\dfrac{\dfrac{\overset{\epsilon_1}{[A \leftarrow A; P'] \Rightarrow B_1} \quad \cdots \quad \overset{\epsilon_n}{[A \leftarrow A; P'] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; P'] \Rightarrow A}\,\text{MP}}{[P] \Rightarrow A}\,\text{EX*}$$

$$\dfrac{\dfrac{\overset{\sigma_1}{[A \leftarrow A; P] \Rightarrow B_1} \quad \cdots \quad \overset{\sigma_n}{[A \leftarrow A; P] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; P] \Rightarrow A}\,\text{MP}}{[P] \Rightarrow A}\,\text{LC}$$

This concludes the completeness proof. □

Observe that the above completeness proof shows that the PFX rule is redundant in any of the three relevant extensions. The inductive step for S+ LC + RC shows that RC is redundant in that calculus. In addition, HP is clearly redundant in S+ EX, as it is subsumed by the Exchange rule.

Combining the above propositions with the observation that **S** + EX corresponds to Classical Logic CL, we find ourselves in the following substructural landscape:

$$\begin{tikzcd}
\mathrm{CL} \equiv \mathbf{S} + \mathrm{EX} \equiv \mathbf{S} + \mathrm{LC} + \mathrm{RC} \equiv \mathbf{S} + \mathrm{W} \arrow[d, dash] \\
\mathbf{S}
\end{tikzcd}$$

3.6 Cut rules

From a proof-theoretical point of view the question of the existence of admissible cut rules for **S** arises, that is, the question whether we can derive sequents by use of lemmas.[1] Let us first investigate the classical notion of cut, which expresses that the cut formula can be replaced by a set of clauses from which it is computable. Its formulation can be found in Figure 1. Classical Cut is not admissible for **S**, as is shown by the following counterexample:

Counterexample Consider the sequent $[A \leftarrow C; C \leftarrow D; C \leftarrow; D \leftarrow D] \Rightarrow A$. While this sequent is not derivable in **S**, in **S** + Classical Cut it has the following derivation (we omit applications of HP and obvious subderivations):

$$\dfrac{\dfrac{[C] \Rightarrow C \qquad [C \leftarrow C; C; D] \Rightarrow D}{[C \leftarrow D; C \leftarrow] \Rightarrow C}\,\mathrm{PFX} \qquad \dfrac{[A \leftarrow A; C; D \leftarrow D] \Rightarrow C}{[A \leftarrow C; C; D \leftarrow D] \Rightarrow A}\,\mathrm{MP}}{[A \leftarrow C; C \leftarrow D; C \leftarrow; D \leftarrow D] \Rightarrow A}\,\mathrm{CC}$$

□

(Likewise, several simple variants of Classical Cut, obtained by permuting P, Y, and Z in the conclusion, can be shown to be inadmissible for **S**.) Therefore, extending **S** with Classical Cut results in a strictly stronger system. The following result, in combination with Proposition 3.5.10, shows that **S** + Classical Cut is intermediary between **S** and any of the extensions with Weakening, Contraction, or Exchange.

3.6.1. PROPOSITION. **S** *+ CC* $\prec$ **S** *+ W.*

PROOF. We first show that **S** + CC $\preceq$ **S** + W. By an easy substitution argument, it can be shown that Classical Cut is sound with respect to the computation semantics defined above for **S** + W. The completeness of **S** + W w.r.t. to successful computations thus gives the required inclusion of **S** + CC in **S** + W.

To prove that inequality holds, observe that

$$\mathbf{S} + \mathrm{W} \vdash [p \leftarrow p; p \leftarrow] \Rightarrow p$$

[1]The procedural counterpart of this question is closely related to the following question concerning operators for combining programs: under which conditions is the computability of an atom proserved after combining the original program with a new module. In the present paper we focus on the proof-theoretic question, rather than on its procedural counterpart, for which work is in progress.

In contrast,

$$\mathbf{S} + \text{Classical Cut} \nvdash [p \leftarrow p; p \leftarrow] \Rightarrow p$$

For suppose that there is a derivation δ in **S** + Classical Cut of $[p \leftarrow p; p \leftarrow] \Rightarrow p$. We show that δ must contain an infinite branch $[P_0] \Rightarrow p$, $[P_1] \Rightarrow p$, ..., with the following properties:

1. For all n, P_n is of the form $p \leftarrow p; P'_n$ for some P'_n ,
2. For all n, the only clauses that occur in P_n are $p \leftarrow$ and $p \leftarrow p$.

Let $P_0 = p \leftarrow p; p \leftarrow$. Then $[P_0] \Rightarrow p$ is the conclusion of δ, and (1) and (2) hold for P_0 . Suppose (1) and (2) hold for P_{n-1}. Then P_{n-1} cannot be an instance of an axiom or the conclusion of an application of HP. So one of the following cases applies:

- P_{n-1} is the conclusion of an application of MP. Then $P_n = P_{n-1}$.
- P_{n-1} is the conclusion of an application of PFX. In this case, take for P_n the right assumption of this application. Then $P_n = P_{n-1}; p \leftarrow$, and clearly P_n are satisfies (1) and (2).
- P_{n-1} is the conclusion of an application of Classical Cut, as follows:

$$\frac{[Y] \Rightarrow C \qquad [T; C \leftarrow; Z] \Rightarrow p}{[T; Y; Z] \Rightarrow p}$$

  where $P_{n-1} = T; Y; Z$.
  By assumption, (2) holds for P_{n-1}, so all the clauses in Y are of the form $p \leftarrow p$ or $p \leftarrow$. Thus C must be p.
  If $T = \emptyset$, then let $P_n = Y$, otherwise let $P_n = T; p \leftarrow; Z$. In both cases, (1) and (2) hold for P_n.

This contradicts the fact that derivations are finite. Thus, b) holds, and we conclude that **S** + CC is strictly weaker than **S** + W. □

Combining these results with those from the previous section, we find ourselves in the following substructural landscape:

$$\begin{tikzcd}
\mathrm{CL} \equiv \mathbf{S} + \mathrm{EX} \equiv \mathbf{S} + \mathrm{LC} + \mathrm{RC} \equiv \mathbf{S} + \mathrm{W} \arrow[d, no head] \\
\mathbf{S} + \mathrm{CC} \arrow[d, no head] \\
\mathbf{S}
\end{tikzcd}$$

In contrast to Classical Cut, the weaker rules Full Context Cut and Partial Context Cut (cf. Figure 2) are sound with respect to **S**. Neither of these two rules is an instance of the other. Both can be shown to be admissible rules for **S**.

3.6.2. PROPOSITION. *Full Context Cut is an admissible rule for* **S**.

PROOF. Let Π and Σ be successful computations for $[P; R] \Rightarrow C$ and $[P; C \leftarrow; R] \Rightarrow A$, respectively. Now observe that, by the presence of the fact-clause $C \leftarrow$ in $P; C \leftarrow; R$, if Σ contains any subcomputation Γ for C, Γ must be successful. Also, we know that C already succeeds from $P; R$. Therefore, substitution of Π for all subcomputations of Σ for C from $P; C; R$ results in a (successful) computation for A from $P; R$. □

As witnessed by the above counterexample, the substitution used in the proof of the above proposition would not give the desired effect for Classical Cut: as success of C from Q does not guarantee success of C from the extended program $P; Q; R$.

Observe that by the constructive nature of the soundness and completeness theorems for **S**, this admissibility result has an algorithmic cut elimination result as a corollary: there exists an effective procedure for the elimination of Full Context Cut. Similar results hold for Partial Context Cut:

3.6.3. PROPOSITION. *Partial Context Cut is an admissible rule for* **S**.

PROOF. Suppose that C succeeds from P and A succeeds from $Q; C \leftarrow; R; Z$, where $P \subseteq Q; R$ and $R_C = \emptyset$. Let Π be the successful computation for A from $Q; C \leftarrow; R; Z$. If C does not occur in Π, then by the inessential difference propoerty, Π is also the successful computation for $Q; R; Z$. So suppose C does occur in Π. By finiteness of computations, C either succeeds or finitely fails from $Q; C \leftarrow; R; Z$. The occurrence of the fact-clause $C \leftarrow$ excludes the latter possibility. Let $(Q; C \leftarrow)_C = \mathcal{C}_1, \ldots, \mathcal{C}_k$. There is an $n \leq k$, such that for every $m < n$, C finitely fails via $\mathcal{C}_m$ from $Q; C \leftarrow; R; Z$, while C succeeds via $\mathcal{C}_n$. We claim that $n < k$. To see this, observe that by the assumption $P \subseteq Q; R$, every branch in the search tree $\mathcal{S}$ for C from P can be permuted and consequently extended to a branch in the search tree $\mathcal{T}$ for C from $Q; C \leftarrow; R; Z$. In particular, any successful branch in $\mathcal{S}$ can be permuted to a successful branch in $\mathcal{T}$. Thus, by the assumption that C succeeds from P, there is a successbranch in $\mathcal{T}$. Moreover, by the assumptions $P \subseteq Q; R$ and $R_C = \emptyset$, this branch must start with resolving C using a C-clause from Q. This proves the claim.
We can now conclude that C succeeds from $Q; C \leftarrow; R; Z$ via a C-clause in Q. Therefore, C also succeeds from $Q; R; Z$, and the successful computations for C from $Q; C \leftarrow; R; Z$ and for C from $Q; R; Z$ are identical. As a consequence, Π, the successful computation for A from $Q; C \leftarrow; R; Z$, is also the successful computation for A from $Q; R; Z$. By soundness and completeness of **S** with respect to the formal computational semantics, the theorem now follows immediately. □

3.6.4. COROLLARY. *There exist effective procedures for the elimination of Full Context Cut and Partial Context Cut.*

It is an open question whether Partial and Full Context Cut are derivable rules for **S**.

3.7 Negation as Failure

Negation as failure is procedurally defined by two rules which express a duality between an atom and its negation:

The literal $\neg A$ succeeds if A fails; it fails if A succeeds. (NAF)

Here failure is interpreted as finite failure. The general depth first search computation mechanism can be extended with the negation as failure rule. As a result we get a computation mechanism that is suitable for querying *normal programs*, that is, programs with negative literals ocurring in the bodies of clauses. In the present section we extend the calculus **S** with rules and an axiom for negation. We show that this extended system $\mathbf{S}_\neg$ correctly formalises general depth first search plus negation as failure. Using the duality between success and failure, we can also extend the notion of formal computation. This extended notion of computations will be used in the correctness proof for $\mathbf{S}_\neg$.

Notation and conventions

- L is a literal if L is either A or $\neg A$, where A is an atom.
- L and K are (sometimes indexed) literals.
- The *atomic part* L^+ of a literal L is defined as follows:

$$\begin{array}{rcl} A^+ & := & A \\ (\neg A)^+ & := & A \end{array}$$

- The *converse* L^- of a literal L is defined as follows:

$$\begin{array}{rcl} A^- & := & \neg A \\ (\neg A)^- & := & A \end{array}$$

- $\mathcal{A}$ is a *(normal) clause* if $\mathcal{A}$ is $A \leftarrow L_1, \ldots, L_n$, where A is a proper atom, and the L_i are literals.
- P is a *(normal) program* if P is a finite list of (normal) clauses.
- Observe that general depth first search is insensitive to the order of literals in the bodies of clauses. Thus a clause $A \leftarrow L_1, \ldots, L_n$ is computationally equivalent with any of the clauses $A \leftarrow L_{p(1)}, \ldots, L_{p(n)}$ if p is a permutation of $\{1, \ldots, n\}$. This justifies the use of the set-notation $A \leftarrow \{L_1, \ldots, L_n\}$ to indicate any of the clauses $A \leftarrow L_{p(1)}, \ldots, L_{p(n)}$.
- To save space in the notation of rules (and derivations), we use the 'sum' $\sum_{i=1}^{n} [P_i] \Rightarrow L_i$ to indicate the list of n assumptions $[P_1] \Rightarrow L_1$, ..., $[P_n] \Rightarrow L_n$.

3.7.1 Axiomatising negation as failure

We axiomatise depth first search with negation as failure in a deductive calculus $\mathbf{S}_\neg$, the rules of which are given in Figure 3.3. The soundness of $\mathbf{S}_\neg$ is almost immediate: Clearly, the axioms AX and the heterogeneous permutation rule HP are sound with respect to depth first search with negation as failure. Also the modus ponens rule MP

$\mathbf{S}_\neg$

Axioms

$[P] \Rightarrow \top$ (AX)

$[P] \Rightarrow \neg A$ if $P_A = \emptyset$ (AX$_\neg$)

Heterogeneous Permutation

$$\frac{[P;\mathcal{C};\mathcal{D};R] \Rightarrow L}{[P;\mathcal{D};\mathcal{C};R] \Rightarrow L}\,\text{HP}$$

where $\mathcal{D}$ and $\mathcal{C}$ are clauses with different heads.

Modus Ponens

$$\frac{[A \leftarrow A;P] \Rightarrow L_1 \quad \ldots \quad [A \leftarrow A;P] \Rightarrow L_n}{[A \leftarrow L_1,\ldots,L_n;P] \Rightarrow A}\,\text{MP}$$

Prefixing

$$\frac{[P] \Rightarrow A \quad \sum_{i=1}^{n} \left[A \leftarrow A;P;L_i^+\right] \Rightarrow L_i^+}{[A \leftarrow L_1,\ldots,L_n;P] \Rightarrow A}\,\text{PFX}$$

$$\frac{[P] \Rightarrow \neg A \quad \sum_{i=1}^{n} \left[A \leftarrow A;P;L_i^+\right] \Rightarrow L_i^+ \quad [A \leftarrow A;P] \Rightarrow L^-}{[A \leftarrow \{L_1,\ldots,L_n,L\};P] \Rightarrow \neg A}\,\text{PFX}_\neg$$

Figure 3.3: The calculus $\mathbf{S}_\neg$.

and the prefixing rule PFX, both formulated for normal clauses, are sound. (Observe that, by the duality of success and finite failure, sequents of the form $[R; L^+] \Rightarrow L^+$ express that the literal L either succeeds or finitely fails from R.) These rules and axioms deal with success of atoms.

The new axiom $\text{AX}_\neg$ and the new rule $\text{PFX}_\neg$ deal with the success of negative literals (i.e., by the duality principle, with the failure of atoms). $\text{AX}_\neg$ expresses the immediate failure of undefined atoms. $\text{PFX}_\neg$ deals with the failure of defined atoms, as follows. Suppose the atom A fails from the program P. Then A will also fail from the program $A \leftarrow L_1, \ldots, L_n; P$ if at least one of the L_i finitely fails from P, while the other body-literals either fail or succeed from P, and none of the L_i has A as a subgoal in the computation from P.

We will more formally investigate soundness and completeness of $\mathbf{S}_\neg$ in the next section.

3.7.2 Computation Semantics

Using the duality principle for failure and success, we can extend the notion of formal computation with the following two clauses, to obtain a proper semantics for $\mathbf{S}_\neg$:

- $\neg\Pi$ is a successful computation for $\neg A$ from P
  iff
  Π is a failing computation for A from P;
- $\neg\Pi$ is a failing computation for $\neg A$ from P
  iff
  Π is a successful computation for A from P.

Example. Consider the program $P = A \leftarrow B, \neg D; B \leftarrow C; D \leftarrow E; C \leftarrow$.
The goal D finitely fails from P, and the associated failing computation is $D[E[\Diamond]]$.
So $\neg D$ succeeds from P, and the associated successful computation is $\neg D[E[\Diamond]]$.
The successful computation for A from P is $A[B[C[\Box]]|\neg D[E[\Diamond]]]$, and A succeeds from P. $\Box$

The extended notion of formal computation inherits the uniqueness property, the finiteness property and the inessential difference property of the original notion. With an obvious extension of the notion of subcomputation, the occurrence property can be formulated. Likewise, the transformation properties are inherited.

We are now in position to state and prove the main theorem of the present section, the correctness theorem for $\mathbf{S}_\neg$.

3.7.1. THEOREM. *The following are equivalent, for normal programs P and literals L:*

$\mathbf{S}_\neg \vdash [P] \Rightarrow L$ (1)
there exists a successful computation for L from P (2)
L succeeds from P via general depth first search + NAF (3)

PROOFSKETCH. The soundness of $\mathbf{S}_\neg$ with respect to the computation semantics is immediate from the definitions. Completeness is proved along the same lines as the completeness of **S**. The axiom for immediate failure, the new prefixing rule, and the modified modus ponens and prefixing rules have enough power to reflect the computational behaviour of negation as failure.

The correctness with respect to general depth first search plus NAF now follows from the fact that the semantical objects formalise search through the relevant search trees. □

With methods analogous to those used in Section 3.4, it can be shown that $\mathbf{S}_\neg$ is decidable.

3.7.3 Substructural properties revisited

It is well-known that the interpretation of negation as finite failure destroys monotonicity. This is reflected in the following difference between **S** and $\mathbf{S}_\neg$. As we have observed, Atomic Monotonicity is admissible for **S**. In contrast, Atomic Monotonicity is not an admissible rule for $\mathbf{S}_\neg$. A simple counterexample is the following: while $\mathbf{S}_\neg \vdash [B \leftarrow] \Rightarrow \neg A$, it is obvious that the sequent $[B \leftarrow; A \leftarrow] \Rightarrow \neg A$ is not derivable in $\mathbf{S}_\neg$.

Another obvious property distinguishes $\mathbf{S}_\neg$ from **S**. We have seen that Classical Logic CL is stronger than **S**. However, it is well-known that in CL no negative information can be derived from normal clauses. In contrast, clearly, negative literals are derivable from normal programs by general depth first search plus negation as failure. Thus, while $\mathbf{S} \prec CL$, $\mathbf{S}_\neg$ is *not* included in CL.

In the context of S, we have seen that Rightward Extension and Rightward Contraction are admissible, but not derivable rules, (Propositions 3.5.6 - 3.5.8). Not surprisingly, these rules are again admissible (but not derivable) in the context of $\mathbf{S}_\neg$. The proofs and counterexamples given in the context of **S** immediately apply, mutatis mutandis, to the context of $\mathbf{S}_\neg$.

Classical Cut CC is not an admissible rule for $\mathbf{S}_\neg$; counterexamples are easily constructed. Full Context Cut FCC is admissible for $\mathbf{S}_\neg$. In contrast, Partial Context Cut PCC is not sound for $\mathbf{S}_\neg$. As a counterexample, observe that the sequents $[C \leftarrow \neg D] \Rightarrow C$ and $[A \leftarrow C; C \leftarrow \neg D; C; D] \Rightarrow A$ are derivable in $\mathbf{S}_\neg$, while $\mathbf{S}_\neg$ does not derive the sequent $[A \leftarrow C; C \leftarrow \neg D; D] \Rightarrow A$.

Negation as failure can be incorporated in the semantics defined for **S** + EX in Section 3.5.2. The extended semantics is correct for the extension of $\mathbf{S}_\neg$ with Exchange and for the extension with Rightward and Leftward Contraction. That is, Exchange and the contraction rules are again equivalent in the context of $\mathbf{S}_\neg$; an addition of these rules corresponds to a shift from depth first search to breadth first search. In contrast, Exchange is not equivalent to Weakening in the context of $\mathbf{S}_\neg$: while the sequent $[B \leftarrow; A \leftarrow] \Rightarrow \neg A$ is derivable in $\mathbf{S}_\neg$ + W, it is not derivable in $\mathbf{S}_\neg$ + EX. It is unclear whether the calculus $\mathbf{S}_\neg$ + W corresponds to any sensible computation procedure.

3.8 Extending the language

The next natural step is to extend of the language of $\mathbf{S}_\neg$ with further connectives and operators. Adding 'logical' connectives in a logic programming context involves a great number of options, that may be distinguished systematically.

First, connectives may be added at various levels:

1. Connectives on goals, inside bodies of clauses. This is the case we will concentrate on in the present section. An interesting feature of an extension with goal connectives is that they act as local implementations of computation rules, on top of the basic processing mechanism.
2. Adding connectives in clause heads is another possibility. The procedural interpretation of goal-connectives in clause heads, however, is not at all obvious, as witnessed by the literature on explicit negation. An interesting option is the use of implication in the head of clauses, to implement the import of modules in the style of [Mil89]. We will, in the present context, not pursue this possibility.
3. At another level, connectives between clauses and on programs themselves are possible. At the level of clauses, procedural connectives could implement various search rules on top of the basic processing mechanism. At the level of programs, procedural versions of module operators would emerge (cf. [BMPT94] and [BT95]). These possibilities remain to be investigated in greater detail.

A potential source of variation in procedural connectives is well-known from substructural and many-valued logics. In a richer computational setting, classical connectives usually 'split' into several natural variants — and even completely new connectives may arise. In the present section, we study several such variants of the classical connectives disjunction and conjunction added to the language of $\mathbf{S}_\neg$ on the level of goals in clauses. We extend the system $\mathbf{S}_\neg$ with (proof theoretic introduction) rules to incorporate these procedural connectives.

3.8.1 Procedural connectives

Computational disjunction (choice) and conjunction (composition) are natural candidates for connectives on goals. We will investigate several of the possible variants of choice and composition, starting with choice. Intuitively, binary disjunction is a choice between the two disjuncts. In a procedural setting, we may call a binary disjunction computable from a program if one of the disjuncts is computable, while failure of a disjunction corresponds to failure of both disjuncts. However, as failure can be caused by either the occurrence of a loop or by finite failure of the search, there is space for a variation in the success conditions as well as the failure conditions for disjunction. We will concentrate on variations in success conditions, and interprete failure of a disjunction as the finite failure of both disjuncts. The simplest possibility is *nondeterministic choice* $A||B$:

> A search for $A||B$ is successful if either the search for A or the search for B is successful; it fails if the searches for both disjuncts end in failure.

This translates immediately into a formal definition of computations:

> $(A||B)P$ is a successful computation for $A||B$ if either AP is a successful computation for A or BP is a successful computation for B; $(A||B)P$ is a failing computation for $A||B$ if P is a tuple $P_A{\star}P_B$, where AP_A is a failing computation for A, and BP_B is a failing computation for B.

Another natural possibility is *left directed choice* A/B, which works as follows. Start with a search for A. Success for A already gives success of A/B, while (finite) failure induces a further search for B. Success of B then gives success for A/B, while (finite) failure of B means failure of A/B. All other possibilities induce a diverging search for A/B. A corresponding notion of formal computation may be defined as above. Of the further interesting options for variants of choice, we want to mention *S-choice* $A{\vee}B$. Success of an S-choice only occurs if one of the disjuncts succeeds and neither loops, while failure occurs if both disjuncts (finitely) fail.

These three alternative procedural disjunctions are obviously related: for instance, observe that success of $A{\vee}B$ implies that of A/B, which again implies success of $A||B$. Considering different failure conditions would again lead to other variants of procedural disjunction, which we will not discuss.

Procedural conjunctions can be introduced via the failure negations of disjunctions. This gives us at least the following options:

CHOICE	*success*			COMPOSITION
NONDETERMINISTIC				PARALLEL
$A\|\|B$	P_A	P_B		$A{\cap}B := \neg(\neg A\|\|\neg B)$
LEFT DIRECTED				LEFT DIRECTED
A/B	P_A	$P_{\neg A}{\star}P_B$		$A{\angle}B := \neg(\neg A/\neg B)$
S-CHOICE				S-COMPOSITION
$A{\vee}B$	$P_A{\star}P_B$	$P_{\neg A}{\star}P_B$	$P_A{\star}P_{\neg B}$	$A{\rhd}B := \neg(\neg A{\vee}\neg B)$

By interpreting $\neg\neg A$ as A (which is in concordance with the interpretation of negation as finite failure), this schema immediately gives the failure conditions of the above three variants of conjunction.

Observe the resemblance to three-valued logic (see, f.i. [Urq86]), where connectives split into different versions as a result of the choices allowed by the truth-value U ('unknown', here related to the possibility of the search going into a loop). This three-valued analogy itself may generate further relevant connectives, (including two clausal negations: a 'weak' and a 'strong' one) that we shall not explore here.

3.8.2 Extending $\mathbf{S}_\neg$ with connectives on goals

An extension of the calculus $\mathbf{S}_\neg$ with the above discussed procedural connectives occurring in bodies of clauses can be obtained in several ways. One possibility is to add side conditions to the original modus ponens and prefixing rules, which reflect the procedural interpretation of the new connectives. The system thus obtained is somewhat laborious, hence we concentrate on another, more classical approach to the procedural goal-connectives, via proof-theoretic introduction rules. The resulting system $\mathbf{S}^+$ is an extension of $\mathbf{S}_\neg$ with introduction rules for double negation and the above discussed procedural versions of disjunction and conjunction. Although several more of the possible procedural connectives can be incorporated in the present setting, we will restrict ourselves to the procedural connectives defined above.

We will use the notions of atom and literal as before, and the new notion of *extended literal* for expressions built up from atoms using the new procedural connectives and the failure negation $\neg$. Extended literals will be written as E, F. We allow iterated occurrences of negation and nesting of the procedural binary connectives in extended literals. Observe that for any extended literal E, $\neg\neg E$ is equivalent to E, by the duality of failure and succes. We adapt the notions of clause and program in the expected way: $\mathcal{A}$ is an (extended) clause if $\mathcal{A}$ is $A \leftarrow E_1, \ldots, E_n$, where the E_i are extended literals and A is a proper atom. A program is a finite list of (extended) clauses.

The introduction rules for the procedural goal-connectives are determined by the corresponding procedural interpretation. The success conditions for the procedural disjunctions $\|$, $/$, and $\vee$, translate into the following introduction rules:

$$\frac{[P] \Rightarrow E}{[P] \Rightarrow E\|F} \qquad \frac{[P] \Rightarrow F}{[P] \Rightarrow E\|F}$$

$$\frac{[P] \Rightarrow E}{[P] \Rightarrow E/F} \qquad \frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow F}{[P] \Rightarrow E/F}$$

$$\frac{[P] \Rightarrow E \quad [P] \Rightarrow F}{[P] \Rightarrow E\vee F} \qquad \frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow F}{[P] \Rightarrow E\vee F} \qquad \frac{[P] \Rightarrow E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow E\vee F}$$

The failure conditions for the various procedural disjunctions (failure of both disjuncts) translate into the following introduction rules for negations:

$$\frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E\|F)} \qquad \frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E/F)}$$

$$\frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E\vee F)}$$

The corresponding introduction rules for the three procedural conjunct ions and their negations are easily derived:

$$\frac{[P] \Rightarrow E \quad [P] \Rightarrow F}{[P] \Rightarrow E \cap F} \qquad \frac{[P] \Rightarrow E \quad [P] \Rightarrow F}{[P] \Rightarrow E \angle F} \qquad \frac{[P] \Rightarrow E \quad [P] \Rightarrow F}{[P] \Rightarrow E \rhd F}$$

$$\frac{[P] \Rightarrow \neg E}{[P] \Rightarrow \neg(E \cap F)} \qquad \frac{[P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E \cap F)}$$

$$\frac{[P] \Rightarrow \neg E}{[P] \Rightarrow \neg(E \angle F)} \qquad \frac{[P] \Rightarrow E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E \angle F)}$$

$$\frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E \rhd F)} \qquad \frac{[P] \Rightarrow \neg E \quad [P] \Rightarrow F}{[P] \Rightarrow \neg(E \rhd F)} \qquad \frac{[P] \Rightarrow E \quad [P] \Rightarrow \neg F}{[P] \Rightarrow \neg(E \rhd F)}$$

In addition, we need the following introduction rule for double negation, which is justified by the duality for success and failure:

$$\frac{[P] \Rightarrow E}{[P] \Rightarrow \neg\neg E}$$

In presence of the above introduction rules for procedural connectives on goals, Modus Ponens and the prefixing rules can prefix clauses in which extended literals occur in the body. Note that the formulation of the prefixing rules used previously requires extended literals to occur in the head of (fact-)clauses, while we wish to restrict the format to extended clauses, that is, clauses with an atomic head. Therefore we will, in the remainder of this chapter, use alternative, equivalent formulations of the prefixing rules. Consider the following assumption of the rule PFX:

$$\sum_{i=1}^{n} \left[A \leftarrow A; P; L_i^+\right] \Rightarrow L_i^+$$

As we have observed in Section 3.7, this assumption expresses that the L_i either succeed or fail on the program $A \leftarrow A; P$. Thus, using the duality between success and finite failure and the option of using iterated negation, this assumption can equivalently be replaced by the following set of assumptions:

$$\sum_{i=1}^{k} [A \leftarrow A; P] \Rightarrow E_i \qquad \sum_{i=k+1}^{n} [A \leftarrow A; P] \Rightarrow \neg E_i \qquad \text{where } 0 \leq k \leq n$$

Thus, the following is an alternative formulation of the rule PFX:

$$\frac{[P] \Rightarrow A \quad \sum_{i=1}^{k} [A \leftarrow A; P] \Rightarrow E_i \quad \sum_{i=k+1}^{n} [A \leftarrow A; P] \Rightarrow \neg E_i}{[A \leftarrow \{E_1, \ldots, E_n\}; P] \Rightarrow A}$$

where $0 \leq k \leq n$.

Observe that the case $k = n$ subsumes the rule MP. Thus we can, without los of strength, eliminate this case[2]. Thus, in the remainder, we will use the following

[2]Observe that this restricts all derivations to normal derivations in the sense of Section 3.4.1.

version of the rule PFX, in which at least one of the body literals of the prefix clause fails:

PFX

$$\frac{[P] \Rightarrow A \quad \sum_{i=1}^{k-1} [A \leftarrow A; P] \Rightarrow E_i \quad \sum_{i=k}^{n} [A \leftarrow A; P] \Rightarrow \neg E_i}{[A \leftarrow \{E_1, \ldots, E_n\}; P] \Rightarrow A}$$

where $1 \leq k \leq n$.

Similarly, we will use the following version of PFX$_\neg$ in the context of $\mathbf{S}^+$:

PFX$_\neg$

$$\frac{[P] \Rightarrow \neg A \quad \sum_{i=1}^{k-1} [A \leftarrow A; P] \Rightarrow E_i \quad \sum_{i=k}^{n} [A \leftarrow A; P] \Rightarrow \neg E_i}{[A \leftarrow \{E_1, \ldots, E_n\}; P] \Rightarrow \neg A}$$

where $1 \leq k \leq n$.

In addition to the above prefixing rules and the introduction rules for the procedural connectives, $\mathbf{S}^+$ has the usual modus ponens rule, the rule for heterogeneous permutation and the axioms of $\mathbf{S}_\neg$(now for extended programs). This completes the description of $\mathbf{S}^+$.

The following proposition shows that for all derivable sequents $[P] \Rightarrow E$, all clauses in P have proper atoms as heads:

3.8.1. PROPOSITION. *If $\mathbf{S}^+ \vdash P \Rightarrow L$, then all clauses in P have atomic heads.*

PROOF. By induction on the depth of derivations. □

The connection between the extended notion of computations (discussed in Section 3.8.1) and the calculus $\mathbf{S}^+$ is as expected:

3.8.2. PROPOSITION. *For all extended literals E and all programs P with only atoms as heads, $\mathbf{S}^+ \vdash P \Rightarrow E$ iff there exists a successful computation for E from P.*

PROOF. By induction, along the same lines as the correctness proof for $\mathbf{S}_\neg$. □

3.8.3 Connectives and selection rules

There is a close connection between the above composition connectives and selection rules. First of all, S-composition inside the body of a clause has the same effect as the basic juxtaposition of body literals in general depth first search. This can be shown on the level of formal computations, as follows. Let P be a program in the language of $\mathbf{S}^+$, let L be a literal. Let P' be the program obtained from P by replacing S-conjunctions with juxtaposition comma's. (For simplicity, assume that the S-conjunctions do not occur under any other connective than S-conjunctions.) Now let Π be a succeeding (failing) $\mathbf{S}^+$-computation for L from P. A corresponding succeeding (failing) computation Π' for L from P' can be obtained by an obvious transformation of Π (replace all subcomputations of the form $(K \rhd L)\Sigma{\star}\Gamma$ by juxtapositions $K\,[\Sigma]\,|L\,[\Gamma]$). Thus, it is seen that S-composition $\rhd$ locally implements the S-computation rule.

Parallel composition in its turn, corresponds to a generalised selection rule which is more liberal in the failure conditions than the S-selection rule: a goal A fails via a clause as soon as one of the body atoms finitely fails.

Similarly, left directed composition locally implements Prolog's left-first selection rule. For example, a search for A via the clause $A \leftarrow B_1 \angle (B_2 \angle B_3)$ mimics computation using Prolog's leftmost selection rule. Observe however, that the underlying search mechanism is basically parallel and does not have the involved backtracking mechanism of standard Prolog. We will more closely investigate the relation beween left directed composition in the context of $\mathbf{S}^+$ and Prolog in the next sections.

Several other selection rules can be implemented by means of other versions of composition connectives. For instance, a rightmost selection rule can be implemented using a right directed version of the procedural composition $\angle$. We will however, in the present context, not persue these possibilities.

3.9 Frugal Prolog

In this section we take a closer look at left directed composition. As we have observed above in Section 3.8.3, left directed composition $\angle$ corresponds to leftmost goal selection. That is, in the context of $\mathbf{S}^+$, left directed composition locally implements Prolog's leftmost selection rule on top of the basic general depth first search mechanism. Still, as we will show, there is a subtle difference between the standard Prolog computation mechanism and the mechanism obtained by specialising the generalised parallel selection rule in general depth first search to leftmost goal selection. This difference is caused by the essentially different backtracking mechanisms employed. We will first investigate the latter mechanism, which, for reasons later to be discussed, we will call frugal Prolog. An indirect characterisation of frugal Prolog can be obtained from $\mathbf{S}^+$. In addition, we will give a calculus $\mathbf{S}_{fPr}$ which directly characterises frugal Prolog. A comparison between frugal and standard Prolog will show that $\mathbf{S}_{fPr}$ is not sound with respect to the standard Prolog computation mechanism. However, for a restricted class of programs, Prolog computation and frugal Prolog computation coincide.

3.9.1 Frugal Prolog

To study the frugal Prolog computation mechanism, we first generalise left directed composition to a version with free arity, as follows:

3.9.1. DEFINITION. (generalised left directed composition)

$$\begin{array}{lcl}\langle L\rangle & := & L\\ \langle L_1, L_2\rangle & := & L_1 \angle L_2\\ \langle L_1, L_2, \ldots, L_n\rangle & := & L_1 \angle \langle L_2, \ldots, L_n\rangle\end{array}$$

□

We will call a clause $\mathcal{A}$ *in Prolog form* if $\mathcal{A}$ is $A \leftarrow \langle L_1, \ldots, L_n \rangle$, where A is a proper atom and the L_i are literals. A program is in Prolog form if all of its clauses are. A sequent $[P] \Rightarrow L$ is in Prolog form if the program P is in Prolog form and L is a literal.

Now suppose a sequent $[P] \Rightarrow L$, which is in Prolog form, is derivable from $\mathbf{S}^+$. Then the literal L succeeds from P via the following computation mechanism:

> **Frugal Prolog computation**
> To prove a goal A from a program, first try the uppermost program clause which has A for its head. A succeeds via a clause $A \leftarrow< B_1, \ldots, B_n >$ if all the B_i succeed successively. A fails finitely via this clause if, for some $i \leq n$, B_i fails finitely, while for all $j < i$, B_j succeeds. In the latter case, try the next lower eligible program clause. The over-all computation procedure involves backtracking to the last choice point. A goal $\neg A$ succeeds iff A finitely fails and vice versa.

Thus we have the following

3.9.2. PROPOSITION. *Let $[P] \Rightarrow L$ be in Prolog form. Then $\mathbf{S}^+ \vdash [P] \Rightarrow L$ iff L succeeds from P via the frugal Prolog computation mechanism.*

While the frugal Prolog computation mechanism is close to the standard Prolog computation mechanism, there is an essential difference. Consider the following program P:

$$\begin{array}{l} A \leftarrow \\ A \leftarrow A \\ B \leftarrow A, C \end{array}$$

A Prolog search for B will diverge: first, A succeeds, but, as C finitely fails, the procedure backtracks and tries to solve A via the second A-clause, on which it goes into loop. A frugal Prolog search however, will establish the finite failure of B, and thus the success of $\neg B$: upon the failure of C, the procedure does not backtrack on the previously established success of A, but will, in the absence of further B-clauses, conclude that B fails. This illustrates the essential difference between frugal and standard Prolog.

Relative to standard Prolog, frugal Prolog search employs less extensive backtracking. More in particular, frugal Prolog does not backtrack on goals for which success has previously been established. This can be interpreted as pruning of the relevant LDNF-trees during frugal Prolog search. In fact, frugal Prolog can be mimicked in standard Prolog, by the use of commit operators on goals.

In the context of predicate logic, backtracking on successful goals has the effect of generating alternative possible answer substitutions, on which the failing subgoal might subsequently succeed. In the propositional case, backtracking on successful goals (which can be interpreted as searching for alternative proofs) has no such effect. Therefore, for propositional programs and goals, frugal Prolog is a reasonable alternative for standard Prolog. In addition, less backtracking means less overhead, so frugal Prolog travels through the search space faster than standard Prolog. Thus

$\mathbf{S}_{fPr}$

AX

$[P] \Rightarrow \top$ where P is in Prolog form.

AX$_\neg$

$[P] \Rightarrow \neg A$ where P is in Prolog form and $P_A = \emptyset$.

Heterogeneous Permutation

$$\frac{[P; \mathcal{C}; \mathcal{D}; R] \Rightarrow A}{[P; \mathcal{D}; \mathcal{C}; R] \Rightarrow A}\ \mathrm{HP} \qquad \text{where } \mathcal{D} \text{ and } \mathcal{C} \text{ are clauses with different heads}$$

Modus Ponens

$$\frac{[A \leftarrow \langle A \rangle; P] \Rightarrow L_1 \quad \ldots \quad [A \leftarrow \langle A \rangle; P] \Rightarrow L_n}{[A \leftarrow \langle L_1, \ldots, L_n \rangle; P] \Rightarrow A}\ \mathrm{MP}$$

Prefixing

$$\frac{[P] \Rightarrow A \quad \sum_{i=1}^{k-1} [A \leftarrow \langle A \rangle; P] \Rightarrow L_i^+ \quad [A \leftarrow \langle A \rangle; P] \Rightarrow (L_k)^-}{[A \leftarrow \langle L_1, \ldots, L_n \rangle; P] \Rightarrow A}\ \mathrm{PFX}$$

where $1 \leq k \leq n$

$$\frac{[P] \Rightarrow \neg A \quad \sum_{i=1}^{k-1} [A \leftarrow \langle A \rangle; P] \Rightarrow L_i^+ \quad [A \leftarrow \langle A \rangle; P] \Rightarrow (L_k)^-}{[A \leftarrow \langle L_1, \ldots, L_n \rangle; P] \Rightarrow \neg A}\ \mathrm{PFX}_\neg$$

where $1 \leq k \leq n$

Figure 3.4: The calculus $\mathbf{S}_{fPr}$.

excection via the frugal Prolog mechanism is more efficient than excecution via standard Prolog. These observations justify a closer look at frugal Prolog.

3.9.2 Characterising frugal Prolog

We can do much better than the implicit characterisation of frugal Prolog in the context of $\mathbf{S}^+$. The calculus $\mathbf{S}_{fPr}$, given in Figure 3.4, correctly axiomatises frugal Prolog, and, unlike $\mathbf{S}^+$, only derives sequents in Prolog form. Instead of introducing left directed composition by proof theoretic introduction rules, the procedural interpretation of left directed composition is used to define appropriate Modus Ponens and Prefixing rules.

We have the following correctness result.

3.9.3. THEOREM. *Let $[P] \Rightarrow L$ be in Prolog form.*
Then $\mathbf{S}^+ \vdash [P] \Rightarrow L$ *iff* $\mathbf{S}_{fP_r} \vdash [P] \Rightarrow L$.

PROOF. From right to left. This is in fact a direct consequence of the considerations that led to the definition of $\mathbf{S}_{fP_r}$. Observe that every instance of the axioms of $\mathbf{S}_{fP_r}$ is an instance of an axiom of $\mathbf{S}^+$. Every instance of MP in $\mathbf{S}_{fP_r}$ is derivable in $\mathbf{S}^+$, using n-1 consecutive applications of the introduction rule for left directed composition followed by an application of MP. Similarly, every instance of PFX (PFX$_\neg$) in $\mathbf{S}_{fP_r}$ is derivable in $\mathbf{S}^+$ by $k-1$ consecutive applications of the introduction rule for the negation of left directed composition followed by an application of the $\mathbf{S}^+$-version of PFX (PFX$_\neg$).

We prove the other direction by induction on the depth of derivations in $\mathbf{S}^+$. Consider an $\mathbf{S}^+$ derivation δ of a sequent in Prolog form. Suppose that the last rule applied was MP. The only instance of this rule of which the conclusion is in Prolog form is the following:

$$\frac{[A \leftarrow A; P] \Rightarrow E}{[A \leftarrow E; P] \Rightarrow A}$$

where $E = \langle L_1, \ldots, L_n \rangle$. Then δ must have the following form:

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ [A \leftarrow A; P] \Rightarrow L_1\end{array} \qquad \begin{array}{c} \dfrac{\begin{array}{c}\vdots \\ [A \leftarrow A; P] \Rightarrow L_{n-1}\end{array} \qquad \begin{array}{c}\vdots \\ [A \leftarrow A; P] \Rightarrow L_n\end{array}}{[A \leftarrow A; P] \Rightarrow \langle L_{n-1}, L_n \rangle} \\ \vdots \\ [A \leftarrow A; P] \Rightarrow \langle L_2, \ldots, L_n \rangle \end{array}}{[A \leftarrow A; P] \Rightarrow \langle L_1, \ldots, L_n \rangle}}{[A \leftarrow \langle L_1, \ldots, L_n \rangle; P] \Rightarrow A}\ \text{MP}$$

The exhibited part of δ can be collapsed into one application of the modus ponens rule of $\mathbf{S}_{fP_r}$.

The prefixing rules of $\mathbf{S}^+$ are analogous. □

A a direct consequence of Proposition 3.9.2 and Theorem 3.9.3, we have the following Theorem.

3.9.4. THEOREM. *Let $[P] \Rightarrow L$ be a sequent in Prolog form. Then L succeeds from P via the frugal Prolog computation mechanism iff* $\mathbf{S}_{fP_r} \vdash [P] \Rightarrow L$.

We have observed above that the difference in computational outcome between standard and frugal Prolog is caused by possible divergence. Therefore, it is to be expected that for programs on which no atom diverges via standard Prolog computation, the two computation mechanisms are equivalent. Indeed, this is true for the class of *left terminating* programs [AP93]. A (propositional) program P is left terminating if all LDNF-derivations on P are finite.

3.9.5. THEOREM. *Let P be a left terminating program. Then $\mathbf{S}_{fP_r}\vdash [P] \Rightarrow L$ iff L succeeds on P via standard Prolog search.*

PROOF. Let P be left terminating. Then every LDNF-derivation from P is finite. Thus every literal either succeeds or finitely fails on P via standard Prolog search. The same is true for computation via frugal Prolog search. In addition, by a simple inductive argument on the depth of formal computations, every successful (failing) $\mathbf{S}^+$ computation for a literal L form P can be extended to a successful (finitely failed) LDNF-tree for $P \cup \{L\}$. □

We will return to the calculus $\mathbf{S}_{fP_r}$ in Section 4.4.

3.10 Notes

- This chapter is based on the ILLC research report CT-94-12 *Gentzen Systems for Logic Programming Styles*, 1994 [Kal94]. The sections 3.1 – 3.7 will appear as [Kal95b] *Gentzen Systems for Logic Programming Styles I: Substructural Aspects* in the Bulletin of the IGPL, in 1995. The remaining sections will appear as the first part of [Kal95c] *Gentzen Systems for Logic Programming Styles II: Logics for Prolog* in the same journal.
- The approach we have used in the present paper, describing search mechanisms for logic programs by means of Gentzen style sequent calculi, was proposed by van Benthem [vB92]. In particular, van Benthem suggested that the calculus **S**, extended with the rules Rightward Contraction and Rightward Extension, is sound and complete w.r.t. the general depth first search mechanism, and gave intuitive soundness and completeness arguments. In the Sections 3.2 and 3.3, we have provided the calculus **S** with a computational semantics and we have formalised the soundness and completeness arguments. In particular, this shows that the rules Rightward Contraction and Rightward Extension are redundant in this setting. Moreover, in Section 3.5, we have shown that these rules are not derivable in **S**.
- The traditional concept of Tarskian consequence operator C_T, which associates with every theory the set of its consequences in T, can be modified to analyse meta-properties of procedural calculi like the above. While the usual consequence operator is an algebraic operator acting on sets of formulas, the corresponding notion for procedural logics acts on equivalence classes of finite logic programs. The equivalence relation is induced by the heterogeneous permutation of clauses. For propositional Horn clause programs P, the (finite) set of goals that succeed on P via a computation mechanism M can be interpreted as a program (i.e., as an equivalence class of programs.) An analysis in the vein of [Mak93] (also cf. [Isr92]) can be conducted for the calculus **S** (cf. [Kal]). The appropriate consequence operator $C^*_{\mathbf{S}}$ can be shown to be idempotent and non-monotonic. In addition, $C^*_{\mathbf{S}}$ does not satisfy inclusion, 'cut', and cautious monotonicity. Soundness (admissibility) of the derivation rules atomic monotonicity, Full Context Cut, and Partial Cut for **S** reflect in (untra-

ditional) properties of $C^*_{\mathbf{S}}$. Unlike most nonmonotonic consequence operators, $C^*_{\mathbf{S}}$ is not supranormal.

- An interesting issue not addressed here is the so-called rule completeness for **S**: an effective characterisation of the derivable and the admissible rules. Such a characterisation would also answer the (obvious) question whether there are other admissible cut rules for **S** besides Partial and Full Context Cut.
- From a procedural point of view, the status of the various cut rules is unclear. The context cut rules can be regarded as program transformation rules. In that case, at least the practicality of FCC is questionable—although an aplication of FCC reduces the size of the program, it possibly increases the time needed to prove the cut formula from the program. The proof of admissibility of PCC shows that for PCC this is not the case.
- It is also an open question whether there is a sensible proof procedure that corresponds to the system **S** + Classical Cut. In fact, this is an instance of the following more general question, reversing the theme of the present chapters: Given a calculus, is there a search mechanism that corresponds to this calculus? Some instances of this question have been answered in this chapter: extensions of the calculus **S** with Weakening, Exchange, and Leftward Extension correspond to a breadth first search mechanism.

Chapter 4

Prolog

None of the calculi discussed in the previous chapter properly reflects standard Prolog computation with leftmost goal selection and top down clause processing. The results and counterexamples discussed in Section 3.9.1 suggest that in order to obtain a procedural calculus that is correct with respect to Prolog computation, a modification or strengthening of the techniques used above is needed. In particular, a proper approach has to account for Prolog's intricate backtracking mechanism.

In the present section, we discuss the Gentzen calculus $\mathbf{S}_{Pr}$, which correctly reflects computation via standard Prolog search. This calculus for Prolog uses, in addition to the usual sequents $[P] \Rightarrow A$, star-sequents $[P] \Rightarrow^* A$. While the sequents $[P] \Rightarrow A$ express success of the atom A from the program P under Prolog computation, the new star-sequents $[P] \Rightarrow^* A$ express, in addition to success of A from P, that the relevant search tree for $P \cup \{A\}$ is finite. The latter property ensures that backtracking on A, in the context of P, does not lead to divergence of the search.

In the present context, we restrict our attention to Prolog computation on *definite* programs, that is, programs in which the bodies are lists of atoms rather than literals. We do however need failure negation $\neg$ in the language of the calculus to express the finite failure of atoms. That is, in addition to the expressions $[P] \Rightarrow A$ and $[P] \Rightarrow^* A$, the calculus $\mathbf{S}_{Pr}$ has expressions $[P] \Rightarrow \neg A$, which correspond to the finite failure (via Prolog search) of the atom A on the (definite) program P. In the context of $\mathbf{S}_{Pr}$, the failure negation $\neg$ is only allowed as a connective on the consequents of sequents.

The calculus $\mathbf{S}_{Pr}$ also derives sequents with consequents Q and $\neg Q$, where Q is a finite list of atoms. That is, the sequents $[P] \Rightarrow Q$, $[P] \Rightarrow^* Q$, and $[P] \Rightarrow \neg Q$ correspond to the success, respectively finite success and finite failure of the composite goal Q from P.

In the present chapter, we will reserve the capital Q to indicate atomic and composite goals, i.e. finite lists of atoms. In addition, in the notation of the $\mathbf{S}_{Pr}$ rules for heterogeneous permutation, we will use L to indicate atoms, composite goals, and their failure negations composite goals.

The relevant search trees for Prolog computation on definite programs are LD-trees, that is, SLD-trees via the leftmost selection rule. We assume familiarity with the concept of SLD-tree ([Llo87]), but because LD-trees will be used in the proof of the correctness theorem for $\mathbf{S}_{Pr}$, we give an explicit definition of LD-trees.

4.0.1. DEFINITION.

- An LD-tree for $P \cup \{B_1, \dots, B_n\}$ is an SLD-tree for $P \cup \{B_1, \dots, B_n\}$ via the leftmost selection rule. The LD-tree for $P \cup \{B_1, \dots, B_n\}$ is unique.
- For leaves C in LD-trees, there are two possibilities:
  - $C = \top$. Then C is marked as success. (Observe that, as before, $\top$ plays the role of the empty query.)
  - The selected atom of C (by leftmost selection, this is the leftmost atom) is an atom which is undefined in the program. Then C is marked as failed.
- A branch b in an LD-tree is a *success branch* if it ends in a leaf marked as success.
- A branch b in an LD-tree is a *failed branch* if it ends in a leaf marked as failed.
- An LD-tree $\mathcal{T}$ is successful for depth-first search, or in short *successful*, if it has a success branch such that all branches left of it end are failed.
- An LD-tree $\mathcal{T}$ is *finitely failed* if ($\mathcal{T}$ is finite and) all its branches are fialed. □

The following lemma expresses that successful (finitely failing) Prolog search corresponds to successful (finitely failed) LD-trees.

4.0.2. LEMMA.
(a) The LD-tree for $P \cup \{B_1, \dots, B_n\}$ is successful iff the goal $B_1, \dots, B_n$ succeeds from P via standard Prolog computation.
(b) The LD-tree for $P \cup \{B_1, \dots, B_n\}$ is finitely failed iff the goal $\{B_1, \dots, B_n\}$ finitely fails from P via standard Prolog computation.

4.1 A Gentzen Calculus for Prolog

The calculus for Prolog, $\mathbf{S}_{Pr}$, of which the complete list of rules can be found in Figure 4.1, contains axioms expressing the immediate success of the empty goal $\top$ and the immediate failure of undefined atoms. The HP rules express Prolog's insensitivity to the relative order of clauses with different heads.

The four rules for composition deal with success and failure of composite goals B, Q where B is an atom and Q a composite goal, while the modus ponens and prefixing rules deal with the success and finite failure of (defined) atoms.

For a motivation of the rules for composition, observe the following. A composite goal B, Q succeeds if (a) the first component B succeeds, (b) the rest of the goal, Q, also succeeds. This is expressed by the first composition rule C1. Under leftmost goal selection, a composite goal can fail finitely in two different ways.

1. The first component B fails. This corresponds to the rule C3.
2. The first component, B, succeeds while the rest of the goal, Q, fails finitely. Due to Prolog's backtracking, after failure of Q has been established, the search backtracks to B and starts a renewed search for B via possible further clauses

Axioms

$[P] \Rightarrow \top$ (AX)

$[P] \Rightarrow^* \top$ (AX*)

$[P] \Rightarrow \neg A$ if $P_A = \emptyset$ ($\mathrm{AX}_\neg$)

HP/HP*

$$\frac{[P;\mathcal{D};\mathcal{C};R] \Rightarrow L}{[P;\mathcal{C};\mathcal{D};R] \Rightarrow L}\,\mathrm{HP} \qquad \frac{[P;\mathcal{D};\mathcal{C};R] \Rightarrow^* L}{[P;\mathcal{C};\mathcal{D};R] \Rightarrow^* L}\,\mathrm{HP}^*$$

if $\mathcal{D}$ and $\mathcal{C}$ are two clauses with different heads.

C1
$$\frac{[P] \Rightarrow B \quad [P] \Rightarrow Q}{[P] \Rightarrow B, Q}$$

C2
$$\frac{[P] \Rightarrow^* B \quad [P] \Rightarrow^* Q}{[P] \Rightarrow^* B, Q}$$

C3
$$\frac{[P] \Rightarrow \neg B}{[P] \Rightarrow \neg(B, Q)}$$

C4
$$\frac{[P] \Rightarrow^* B \quad [P] \Rightarrow \neg Q}{[P] \Rightarrow \neg(B, Q)}$$

MP
$$\frac{[A \leftarrow A; P] \Rightarrow Q}{[A \leftarrow Q; P] \Rightarrow A}$$

PFX₁
$$\frac{[P] \Rightarrow A \qquad [A \leftarrow A; P] \Rightarrow \neg Q}{[A \leftarrow Q; P] \Rightarrow A}$$

MP#
$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow^* Q}{[A \leftarrow Q; P] \Rightarrow^* A}$$

PFX₂
$$\frac{[P] \Rightarrow \neg A \qquad [A \leftarrow A; P] \Rightarrow \neg Q}{[A \leftarrow Q; P] \Rightarrow \neg A}$$

MP*
$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow^* Q}{[A \leftarrow Q; P] \Rightarrow^* A}$$

PFX₃
$$\frac{[P] \Rightarrow^* A \qquad [A \leftarrow A; P] \Rightarrow \neg Q}{[A \leftarrow Q; P] \Rightarrow^* A}$$

Figure 4.1: The rules of $\mathbf{S}_{Pr}$

for B. Divergence of this renewed search for B would imply divergence of the search for B, Q instead of finite failure. Thus B must have a finite, successful search tree. This is expressed with a starsequent $[P] \Rightarrow^* B$. This case corresponds to the rule C4.

Further, the rule C2 deals with the finite success of a composite query.

Observe that the composition connective defined with these rules is a version of the left-directed composition $\angle$ defined in Section 3.8 and generalised in Section 3.9.

The modus ponens and prefixing rules of $\mathbf{S}_{Pr}$ deal with the success and finite failure of *defined atoms* A from a program R. It will be useful to assume (without loss of generality, by heterogeneous permutation) that

$$R = A \leftarrow Q; P.$$

Let $\mathcal{T}$ be the LD-tree for $R \cup \{A\}$.

We distinguish two cases.

1. The atom A succeeds on the program R. (That is, $\mathcal{T}$ is successful.) Two subcases can be distinguished.
   (a) A succeeds via the leftmost A-clause in R. That is, Q succeeds while during the search for Q, A is not encountered as a subgoal. This corresponds to a unary version of the usual modus ponens rule MP.
   (b) A fails via the leftmost A-clause $A \leftarrow Q$, but succeeds via some further A-clause in P. That is, first the search for Q fails, and A is not encountered as a subgoal during that search, then A succeeds from P. The prefixing rule corresponding to this case is PFX_1.
2. The atom A fails finitely from R. That is, A fails via the leftmost A-clause $A \leftarrow Q$, and subsequently also fails via the further A-clauses in P. As before, the latter condition corresponds to an assumption $[P] \Rightarrow \neg A$. The first condition corresponds, as in the case 1(b), to the unary prefixing assumption for the failure of Q. The resulting prefixing rule is PFX_2.

Furthermore, we need additional rules and axioms to derive star-sequents. Keeping in mind the intended meaning of the star-sequents, the rules deriving star-sequents cover the various cases in which the relevant LD-tree $\mathcal{T}$ is finite and successful. $\mathbf{S}_{Pr}$ has a star-version of the axiom for the empty goal, and a star version HP* of the rule for heterogeneous permutation. In analogy to the above motivation for the rules MP, PFX_1, and PFX_2, which have as a consequent a non-star sequent $[R] \Rightarrow A$, we can distinguish the following two cases.

1. A succeeds via the leftmost A-clause. This case corresponds to a strengthening of the modus ponens rule.
   First, the subtree generated by the composite query Q is finite and successful. This is expressed by a starversion of the assumption of the modus ponens rule MP: $[A \leftarrow A; P] \Rightarrow^* Q$.
   For the rest of $\mathcal{T}$ (corresponding to possible search for A via further A-clauses in P), two cases can be distinguished: it either contains a success branch or it

contains only failed branches.
The first case corresponds to success of A from P (with a finite LD-tree), reflected in an extra assumption $[P] \Rightarrow^* A$. This results in the rule MP*.
The second case corresponds to finite failure of A from P, and is reflected in an extra assumption $[P] \Rightarrow \neg A$. This results in the rule MP#.

2. A fails via the leftmost A-clause $A \leftarrow Q$, but succeeds via some further A-clause in P. The first condition translates into an assumption $[A \leftarrow A; P] \Rightarrow \neg Q$, as in the case 1(b). The second condition translates into the assumption $[P] \Rightarrow^* A$. This results in the rule PFX_3.

This completes the description and motivation of the rules of the calculus $\mathbf{S}_{Pr}$.

In the next section, we will formalise the above intuitive correctness arguments, and provide a formal proof of the correctness of $\mathbf{S}_{Pr}$ with respect to Prolog computation.

The following simple property of $\mathbf{S}_{Pr}$ can be easily proved by induction on the depth of derivations.

4.1.1. PROPOSITION.
For all queries Q, if $\mathbf{S}_{Pr} \vdash [P] \Rightarrow^* Q$ then $\mathbf{S}_{Pr} \vdash [P] \Rightarrow Q$.

4.2 Soundness and Completeness

In every LD-tree $\mathcal{T}$, we can distinguish a minimal part that witnesses the success, finite failure, or divergence of a Prolog depth first search on $\mathcal{T}$. These relevant parts of LD-trees, the search paths, will play a role in the soundness and completeness proof which is comparable to the role played by computations in the previous soundness and completeness results.

4.2.1. DEFINITION. Let $\mathcal{T}$ be an LD-tree. The *search path* of $\mathcal{T}$ is the following part of $\mathcal{T}$:
(1) If $\mathcal{T}$ is successful, then the search path consists of the leftmost success branch and all (failed failed) branches to the left of it.
(2) If $\mathcal{T}$ is finitely failed, then the search path is $\mathcal{T}$ itself.
(3) Otherwise, the search path consists of the leftmost infinite branch and all (failed failed) branches left of it. □

Observe that the search path of an LD-tree $\mathcal{T}$ is finite if and only if $\mathcal{T}$ is successful or finitely failed.

The completeness proof will proceed by induction on the weight of LD-trees. The weight is the tuple of the number of nodes of the tree proper and the number of nodes of its search path.

4.2.2. DEFINITION. Let $\mathcal{T}$ be an LD-tree. The *weight* of $\mathcal{T}$, $w(\mathcal{T})$, is the tuple

$$w(\mathcal{T}) = \langle \sharp(\mathcal{T}), sw(\mathcal{T}) \rangle,$$

where

$\sharp(\mathcal{T})$ is the number (i.e. the cardinality) of nodes of $\mathcal{T}$;
$sw(\mathcal{T})$, the *searchweight* of $\mathcal{T}$, is the number of nodes in its search path. □

Observe that both components of the weight can be infinite (indicated as ∞). However, in the completeness proof below, we will only be concerned with the weight of successful and finitely failed LD-trees, that is, trees of which the searchweight is finite. Thus, the range of the weight function on the relevant domain is $(\omega + 1) \times \omega$. On the weight of LD-trees we impose a lexicographical order $<_o$, as follows:

4.2.3. DEFINITION. $\langle a, b\rangle <_o \langle c, d\rangle$ iff $a < c$, or $a = c \;\&\; b < d$. □

The order type of this order on the intended range of the weight function is $(\omega + 1)\omega$.

We will now state some useful lemmas, which establish relations between the LD-tree for a composite goal, and the LD-trees for the components. We omit the proofs.

4.2.4. LEMMA. *Let $\mathcal{T}$ be the LD-tree for $P \cup \{B_1, \ldots, B_n\}$, where $n > 1$, and let for $i \in [1, n]$, $\mathcal{T}_i$ be the LD-tree for $P \cup \{B_i\}$. Then*
(a) If $\mathcal{T}$ is successful, then, for $i \in [1, n]$, $\mathcal{T}_i$ is successful and $w(\mathcal{T}_i) <_o w(\mathcal{T})$.
(b) If, for all $i \in [1, n]$, $\mathcal{T}_i$ is successful, then $\mathcal{T}$ is successful.

4.2.5. LEMMA. *Under the conditions of Lemma 4.2.4, the following hold:*
(a) If $\mathcal{T}$ is successful and finite, then, for $i \in [1, n]$, $\mathcal{T}_i$ is successful and finite, and $w(\mathcal{T}_i) <_o w(\mathcal{T})$.
(b) If $\mathcal{T}_i$ is successful and finite, for all $i \in [1, n]$, then $\mathcal{T}$ is successful and finite.

4.2.6. LEMMA. *Under the assumptions of Lemma 4.2.4, the following hold:*
$\mathcal{T}$ is finitely failed iff there is a $k \leq n$ (say $k_{\mathcal{T}}$) such that
(1) $\mathcal{T}_k$ is finitely failed, and
(2) for all $i < k$, $\mathcal{T}_i$ is successful and finite.
If $\mathcal{T}$ is finitely failed, then $w(\mathcal{T}_i) <_o w(\mathcal{T})$, for $i \leq k_{\mathcal{T}}$.

The following lemmas are the proper equivalents of the inessential difference and occurrence properties of formal computations (cf. Section 3.2).

4.2.7. LEMMA. *Let P and R be programs which only differ with respect to clauses with head A. Let Q be a query. Let $\mathcal{T}$ be the LD-tree for $P \cup \{Q\}$, and let $\mathcal{S}$ be the LD-tree for $R \cup \{Q\}$. Then the following hold.*

1. *Let b be a branch in $\mathcal{T}$ such that A is never selected on b. Then b is also a branch in $\mathcal{S}$.*
2. *If A is never selected in the search path of $\mathcal{T}$, then the search path of $\mathcal{T}$ is identical to the search path of $\mathcal{S}$.*
3. *If A is never selected in $\mathcal{T}$ then $\mathcal{S}$ is identical to $\mathcal{T}$.*

4.2.8. LEMMA. *Let Q be a query, and let $\mathcal{T}$ be the LD-tree for $A \leftarrow A; P \cup \{Q\}$.*

1. *If $\mathcal{T}$ is finite, then A is never selected in $\mathcal{T}$.*
2. *If $\mathcal{T}$ is successful, then A is never selected in the search path of $\mathcal{T}$.*

4.2.9. LEMMA. *Let $\mathcal{T}$ be the LD-tree for $P \cup \{A\}$.*

1. *If $\mathcal{T}$ is finite, then A is never selected in $\mathcal{T}$ except in the root of $\mathcal{T}$.*
2. *If $\mathcal{T}$ is successful, then A is never selected on the search path of $\mathcal{T}$, except in the root of $\mathcal{T}$.*

We are now in position to state and prove the main lemma of this section. The correctness result for $\mathbf{S}_{Pr}$ is an immediate consequence.

4.2.10. LEMMA.

1. $\mathbf{S}_{Pr} \vdash [P] \Rightarrow Q$ *iff the LD-tree for $P \cup \{Q\}$ is successful.*
2. $\mathbf{S}_{Pr} \vdash [P] \Rightarrow^* Q$ *iff the LD-tree for $P \cup \{Q\}$ is finite and successful.*
3. $\mathbf{S}_{Pr} \vdash [P] \Rightarrow \neg Q$ *iff the LD-tree for $P \cup \{Q\}$ is finitely failed.*

PROOF. We first prove the soundness side of the claim (left-to-right), then the completeness side (right-to-left).

SOUNDNESS
(By induction on the depth of derivations.) The base case, for derivations that are axioms, is trivial.
(IH) Suppose the left-to-right side of the claim holds for derivations of depth $< m$. Let δ be a derivation of depth m. We distinguish several cases, according to the last rule applied in δ.

Soundness of the rules for heterogeneous permutation follows from the fact that LD-trees are insensitive to heterogeneous permutation of clauses.

Soundness of the composition rules C1–C4 follows from the lemmas 4.2.4 – 4.2.6.

Suppose that the last rule applied in δ is MP. Then δ has the following form:

$$\frac{\begin{array}{c}\vdots\\ [A \leftarrow A; P] \Rightarrow Q\end{array}}{[A \leftarrow Q; P] \Rightarrow A}$$

By the inductive hypothesis, the LD-tree for $A \leftarrow A\,;\ P \cup \{Q\}$ is successful. Using the Lemmas 4.2.8(2) and 4.2.7(2), we infer that the LD-tree for $A \leftarrow Q\,;\ P \cup \{Q\}$ is successful. Thus the LD-tree for $A \leftarrow Q; P \cup \{A\}$ is successful.

Suppose that the last rule applied in δ is MP#. Then δ has the following form:

$$\frac{\begin{array}{cc}\vdots & \vdots\\ [P] \Rightarrow \neg A & [A \leftarrow A; P] \Rightarrow^* Q\end{array}}{[A \leftarrow Q\,;\ P] \Rightarrow^* A}$$

Let $\mathcal{T}$ be the LD-tree for $A \leftarrow Q\,;\ P \cup \{A\}$. We will show that $\mathcal{T}$ is successful and finite.
Let $\mathcal{L}$ be the LD-tree for $A \leftarrow Q\,;\ P \cup \{Q\}$. By the inductive hypothesis, the LD-tree

for $A \leftarrow A\,;\, P \cup \{Q\}$ is finite and successful. Thus, from the Lemmas 4.2.8(1) and 4.2.7(3), it follows that $\mathcal{L}$ is successful and finite.
Let $\mathcal{R}$ be the LD-tree for $P \cup \{A\}$. By the inductive hypothesis, $\mathcal{R}$ is finitely failed.
It remains to show that $\mathcal{T}$ is finite.
Let $b = b_1, b_2, \ldots$, with $b_1 = A$, be a branch of $\mathcal{T}$. We will show that b is either a branch of $\mathcal{L}$ or a branch of $\mathcal{R}$, and thus finite. Suppose $b_2 = Q$. Then $b_2, b_3, \ldots$ is a branch in $\mathcal{L}$. Thus b is finite. So suppose $b_2 \neq Q$. If A is never selected on $b_2, b_3, \ldots$, then b is (by Lemma 4.2.7(1)) a branch in $\mathcal{R}$, and thus b is finite in that case. So suppose A is selected in $b_2, b_3, \ldots$. Let $k > 1$ be minimal such that A is selected in b_k. Observe that $b_1, \ldots, b_k$ must be an initial segment of a branch in $\mathcal{R}$. In particular, A is selected in a node of $\mathcal{R}$ different from the root. But that contradicts Lemma 4.2.9(1). Thus A cannot be selected on $b_2, b_3, \ldots$. We conclude that b is finite. Thus $\mathcal{T}$ is finite.

Suppose that the last rule applied in δ is PFX$_1$. Then δ has the following form:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ [P] \Rightarrow A & [A \leftarrow A; P] \Rightarrow \neg Q \end{array}}{[A \leftarrow Q\,;\, P] \Rightarrow A}$$

By the inductive hypothesis and the Lemmas 4.2.8(1) and 4.2.7(3), the LD-tree for $A \leftarrow Q; P \cup \{Q\}$ is finitely failed. By inductive hypothesis the LD-tree for $P \cup \{A\}$ is successful. The Lemmas 4.2.9(2) and 4.2.7 can be applied to show that the LD-tree for $A \leftarrow Q; P \cup \{A\}$ is successful.

The other cases go via similar arguments.
This concludes the proof of the soundness side of the claim.

COMPLETENESS
By induction on the weight of LD-trees. Observe that while the search path of a successful or finitely failed LD-tree is finite, the number of nodes in a successful tree might be infinite. As a result, we need transfinite induction.
The base case, $w(\mathcal{T}) = \langle 1, 1\rangle$, corresponds to the axioms for immediate failure and to the axioms for immediate success of the empty goal.

(IH) Suppose that the completeness side of the claim holds for LD-trees $\mathcal{T}$ with $w(\mathcal{T}) <_o \langle k, m\rangle$.

Let $\mathcal{T}$ be the successful or finitely failed LD-tree with $w(\mathcal{T}) = \langle k, m\rangle$. We distinguish two cases: (A) $\mathcal{T}$ is an LD-tree for a composite query, and (B) $\mathcal{T}$ is an LD-tree for an atomic query.

(A) Let $\mathcal{T}$ be the successful or finitely failed LD-tree for a composite query, i.e. for $P \cup \{B, Q\}$.
If $\mathcal{T}$ is successful, we infer by Lemma 4.2.4 that the respective search trees for $P \cup \{B\}$

and $P \cup \{Q\}$ are successful, and that their respective weights are smaller than $w(\mathcal{T})$. Thus the inductive hypothesis applies, and the sequents $[P] \Rightarrow B$ and $[P] \Rightarrow Q$ are derivable. By an application of the composition rule C1 we infer that $[P] \Rightarrow B, Q$ is derivable.
If $\mathcal{T}$ is successful and finite, we similarly infer, by Lemma 4.2.5, the inductive hypothesis, and an application of C2, that $[P] \Rightarrow^* B, Q$ is derivable.
Similarly, if $\mathcal{T}$ is finitely failed, Lemma 4.2.6 applies, and either C3 or C4 is applicable to infer that $[P] \Rightarrow \neg(B, Q)$ is derivable.
This exhausts the possibilities in the case $\mathcal{T}$ is an LD-tree for a composite query.

(B) Let $\mathcal{T}$ be a finitely failed or successful LD-tree for $P \cup \{A\}$, with $w(\mathcal{T}) = \langle k, m \rangle$. If $P_A = \emptyset$ or A is $\top$, we are in the base case. So, without loss of generality, we can assume, by the HP and HP* rule that $P = A \leftarrow Q; R$ for some R. Thus, the root A of $\mathcal{T}$ has for its leftmost child the node Q.

We will split $\mathcal{T}$ in two relevant parts $\mathcal{T}_L$ and $\mathcal{T}_R$, as follows:

$\mathcal{T}_L$ is the subtree of $\mathcal{T}$ generated by the leftmost immediate descendant of the root.
$\mathcal{T}_R$ is the result of deleting $\mathcal{T}_L$ from $\mathcal{T}$.

Note that $\mathcal{T}_L$ is the LD-tree for $R \cup \{Q\}$. Observe that $\mathcal{T}_R$ is not necessarily an LD-tree.

In addition, we need the following two LD-trees $\mathcal{L}$ and $\mathcal{R}$, which are closely related to $\mathcal{T}_L$ and $\mathcal{T}_R$:

$\mathcal{L}$ is the LD-tree for $A \leftarrow A\,; R \cup \{Q\}$;

$\mathcal{R}$ is the LD-tree for $R \cup \{A\}$.

We will use the following observations:

- By assumption, $\mathcal{T}$ is successful or finitely failed. Therefore, by Lemma 4.2.9, A is never selected on the search path of $\mathcal{T}$, except in the root. The search path of $\mathcal{T}$ extends the search path of $\mathcal{T}_L$. Consequently, A is never selected on the search path of $\mathcal{T}_L$. Therefore, by Lemma 4.2.7, the search paths of $\mathcal{T}_L$ and $\mathcal{L}$ are identical. As a result

  (1) $\mathrm{sw}(\mathcal{L}) = \mathrm{sw}(\mathcal{T}_L)$
  (2) $\mathcal{L}$ is finitely failed iff $\mathcal{T}_L$ is finitely failed
  (3) $\mathcal{L}$ is successful iff $\mathcal{T}_L$ is successful.

- Suppose that $\mathcal{T}$ is finite. Then A is never selected on $\mathcal{T}$, except in the root (Lemma 4.2.9(1)). Thus A is never selected in $\mathcal{T}_L$ and A is only selected in the root of $\mathcal{T}_R$. As a consequence, using Lemma 4.2.7,

  (4) $\mathcal{R}$ is identical to $\mathcal{T}_R$
  (5) $\mathcal{L}$ is identical to $\mathcal{T}_L$.

Additionally, from finiteness of $\mathcal{T}$, it follows that $\sharp(\mathcal{T}_R) < \sharp(\mathcal{T})$ and $\sharp(\mathcal{T}_L) < \sharp(\mathcal{T})$. Thus the following hold:

$$\begin{aligned}&(6)\quad w(\mathcal{R}) <_o w(\mathcal{T})\\ &(7)\quad w(\mathcal{L}) <_o w(\mathcal{T}).\end{aligned}$$

(Note that in this case, if $\mathcal{T}$ is finite and $\mathcal{T}_R$is successful, the search weight of $\mathcal{T}_R$ may be larger than the search weight of $\mathcal{T}$. Because of that, induction on the search weight is not appropriate.)

- Suppose $\mathcal{T}$ is not finite. We distinguish the following cases:
  - $\mathcal{T}_L$ is finitely failed. Then, by Lemma 4.2.7, $\mathcal{T}_L$ is identical with $\mathcal{L}$. Thus $\sharp(\mathcal{L}) < \sharp(\mathcal{T}) = \infty$. As a consequence,

    $$(8)\quad w(\mathcal{L}) <_o w(\mathcal{T}).$$

    Also, by the Lemmas 4.2.7 and 4.2.9, the search path of $\mathcal{R}$ is equal to the search path of $\mathcal{T}$ restricted to $\mathcal{T}_R$. Thus $\mathcal{R}$ is successful and $sw(\mathcal{R}) < sw(\mathcal{R}) + sw(\mathcal{L}) = sw(\mathcal{T})$. Obviously $\sharp(\mathcal{R}) \leq \sharp(\mathcal{T}) = \infty$. So

    $$(9)\quad w(\mathcal{R}) <_o w(\mathcal{T}).$$

  - $\mathcal{T}_L$ is successful. Obviously, $\sharp(\mathcal{L}) \leq \sharp(\mathcal{T}) = \infty$. Also, $sw(\mathcal{T}_L) + 1 = sw(\mathcal{T})$. So, by (1) above, it follows that (8) also holds in this case.

Using, the above observations, we distinguish the following cases:

- $\mathcal{T}$ is successful but not finite. Using (2) and (3), we distinguish two subcases: $\mathcal{L}$ is successful or $\mathcal{L}$ is finitely failed. In the latter case, as we have seen above, $\mathcal{R}$ is successful.
  - $\mathcal{L}$ is successful. By (8) and the inductive hypothesis, we conclude that there is an $\mathbf{S}_{Pr}$-derivation of the sequent

    $$[A \leftarrow A\,;\, R] \Rightarrow Q \qquad \text{(a)}$$

    An application of MP gives a derivation of

    $$[P] \Rightarrow A \qquad \text{(I)}$$

  - $\mathcal{L}$ is finitely failed, and $\mathcal{R}$ is successful. By (8) and the inductive hypothesis, the following sequent is derivable in $\mathbf{S}_{Pr}$:

    $$[A \leftarrow A; R] \Rightarrow \neg Q \qquad \text{(b)}$$

    Also, by (9) and the inductive hypothesis, there is a derivation of

    $$[R] \Rightarrow A \qquad \text{(d)}$$

    An application of PFX_1 gives a derivation of (I).

- $\mathcal{T}$ is successful and finite. Using (4) and (5), we can distinguish three subcases:
  - $\mathcal{L}$ is successful (and finite) and $\mathcal{R}$ is (finitely) failed. By (6) and the inductive hypothesis, there is a derivation of

    $$[R] \Rightarrow \neg A \tag{e}$$

    Also, by (8) and inductive hypothesis, there is a derivation of

    $$[A \leftarrow A\,;\, R] \Rightarrow^{*} Q. \tag{f}$$

    An application of MP# gives a derivation of

    $$[P] \Rightarrow^{*} A \tag{II}$$

  - $\mathcal{L}$ is finitely failed and $\mathcal{R}$ is finite and successful. Again, we get derivability of the sequent (b). Also, by (6) and the inductive hypothesis, there is a derivation of the sequent

    $$[R] \Rightarrow^{*} A \tag{g}$$

    Thus by an application of PFX_3, we can infer derivability of the sequent (II).
  - Both $\mathcal{L}$ and $\mathcal{R}$ are finite and successful. By an application of MP* we can conclude derivability of (II).
- $\mathcal{T}$ is finitely failed. Then, by (4) and (5), both $\mathcal{L}$ and $\mathcal{R}$ are finitely failed. Again, by (6) and the inductive hypothesis, we infer derivability of the sequent (e). By (7) and the inductive hypothesis we infer that (b) is derivable. Thus, by using PFX_2, we conclude that the sequent

  $$[P] \Rightarrow \neg A \tag{III}$$

  is derivable.

This completes the proof of the correctness lemma. □

As a result, we have correctness of $\mathbf{S}_{Pr}$.

4.2.11. THEOREM. *For all queries Q,*
$\mathbf{S}_{Pr} \vdash [P] \Rightarrow Q$ *iff Q succeeds from P via standard Prolog computation.*
$\mathbf{S}_{Pr} \vdash [P] \Rightarrow \neg Q$ *iff Q finitely fails from P via standard Prolog computation.*

PROOF. The theorem immediately follows from the above correctness lemma and Lemma 4.0.2. □

4.3 Variants and Extensions

A variant of $\mathbf{S}_{P_r}$, which has only sequents with atomic consequents and their negations, is obtained by omitting the composition rules C1 – C4 and incorporating the composition in the modus ponens and prefixing rules. This alternative calculus $\mathbf{S}^-_{P_r}$, of which the modus ponens and prefixing rules are given in Figure 4.2, is less expressive than $\mathbf{S}_{P_r}$, as it does not deal explicitly with success and failure of composite queries. The equivalence of $\mathbf{S}_{P_r}$ and $\mathbf{S}^-_{P_r}$ and thus the correctness of $\mathbf{S}^-_{P_r}$ w.r.t. Prolog computation, can be proved along the same lines as Theorem 3.9.3.

Using the alternative calculus $\mathbf{S}^-_{P_r}$, it is now simple to establish the *completeness* of the frugal Prolog computation mechanism with respect to Prolog computation. The proof, which we omit, uses induction on $\mathbf{S}^-_{P_r}$-derivations, the equivalence of $\mathbf{S}_{P_r}$ and $\mathbf{S}^-_{P_r}$, and the above Proposition 4.1.1.

4.3.1. THEOREM. *If* $\mathbf{S}_{P_r} \vdash [P] \Rightarrow L$ *then* $\mathbf{S}_{fP_r} \vdash [P] \Rightarrow L$.

As we have observed in the previous section, $\mathbf{S}_{fP_r}$ is not *sound* with respect to Prolog computation. That is, the converse of Theorem 4.3.1 does not hold.

The original calculus $\mathbf{S}_{P_r}$ is better suited for extensions with other procedural connectives on goals. For the calculus $\mathbf{S}$, we have studied these possibilities to some extent in Section 3.8. A similar analysis could now be executed in the context of $\mathbf{S}_{P_r}$. We will not pursue this possibility here.

The calculus $\mathbf{S}_{P_r}$ can be easily generalised for *normal programs.* A completeness proof for this extended calculus essentially follows the same lines as the above proof. However, the relevant search trees for normal programs are LDNF-trees rather than LD-trees, which complicates the correctness proof for that more general case. We omit further details.

4.4 Linear Logic

Cerrito [Cer92] uses the framework of Linear Logic for a characterisation of Prolog computation on propositional normal programs. We summarise her approach and main result:

> With each normal, propositional program P a set of one-sided linear sequents $\mathbf{LT}_P$ is associated, expressing the success and failure conditions of the atoms occurring in P. An atom A Prolog-succeeds from P if and only if the sequent $\vdash A$ is derivable from the theory $\mathbf{LT}_P$ in Linear Logic, and the sequent $\vdash A^\perp$ is derivable in Linear Logic from $\mathbf{LT}_P$ if and only if A finitely fails from P via Prolog computation. By the duality of success and finite failure, this result generalises from atoms to literals.

The soundness part of this result is essentially flawed. As a counterexample, consider the following program P:

$$\begin{array}{l} p \leftarrow \\ p \leftarrow p \\ r \leftarrow p, q \end{array}$$

MP

$$\frac{[A \leftarrow A; P] \Rightarrow B_1 \quad \dots \quad [A \leftarrow A; P] \Rightarrow B_n}{[A \leftarrow B_1, \dots, B_n; P] \Rightarrow A}$$

MP#

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \quad \dots \quad [A \leftarrow A; P] \Rightarrow^* B_n}{[A \leftarrow B_1, \dots, B_n; P] \Rightarrow^* A}$$

MP*

$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \quad \dots \quad [A \leftarrow A; P] \Rightarrow^* B_n}{[A \leftarrow B_1, \dots, B_n; P] \Rightarrow^* A}$$

PFX$_1$

$$\frac{[P] \Rightarrow A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \ \dots \ [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow B_1, \dots, B_n, D, C_1, \dots, C_m; P] \Rightarrow A}$$

PFX$_2$

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \ \dots \ [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow B_1, \dots, B_n, D, C_1, \dots, C_m; P] \Rightarrow \neg A}$$

PFX$_3$

$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \ \dots \ [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow B_1, \dots, B_n, D, C_1, \dots, C_m; P] \Rightarrow^* A}$$

Figure 4.2: The modus ponens and prefixing rules of $\mathbf{S}^-_{P_r}$

The associated theory $\mathbf{LT}_P$ contains the sequents

$$\begin{gathered}\vdash p\\ \vdash q^{\perp}\\ \vdash p^{\perp}, q, r^{\perp}\end{gathered}$$

Thus the sequent $\vdash r^{\perp}$ is a linear consequent of $\mathbf{LT}_P$. According to soundness, this would correspond to the finite failure of r under Prolog computation. However, a Prolog search for r will go into loop. Thus the theories $\mathbf{LT}_P$ do not faithfully describe the outcome of Prolog computation.

The success and failure conditions expressed in $\mathbf{LT}_P$ do not account for the standard Prolog backtracking mechanism. In fact, they correspond to the alternative frugal backtracking mechanism discussed in Section 3.9.1:

4.4.1. THEOREM. *The sequent $\vdash A$ is a linear consequent of the theory $\mathbf{LT}_P$ iff $\mathbf{S}_{fPr} \vdash [P] \Rightarrow A$. The sequent $\vdash A^{\perp}$ is a linear consequent of the theory $\mathbf{LT}_P$ iff $\mathbf{S}_{fPr} \vdash [P] \Rightarrow \neg A$.*

PROOF. There is a strong correspondence between successful and failing computations for a program P and sequents derivable from the theory $\mathbf{LT}_P$. This correspondence can be used to obtain an inductive proof. □

The essentially substructural character of procedural logic seems to be illustrated by Cerrito's application of Linear Logic to obtain a logic characterising a Prolog variant. However, the role of Linear Logic in Cerrito's approach is negligible. This can be seen by the following arguments:

1. Linear negation $^{\perp}$ is the only (linear) connective that occurs in the theories $\mathbf{LT}_P$ and in the consequences of interest in this context, $\vdash A$ and $\vdash A^{\perp}$. By the subformula property for Linear Logic, this implies that the full strength of Linear Logic is not needed—the correctness result (that is, the correctness result with respect to frugal Prolog) is already true for the subsystem consisting of the linear axioms, the cut rule, and the exchange rule.
2. A careful analysis of the completeness proof however shows that both the linear axioms and the exchange rule are redundant. Thus the correctness result can be obtained for a system that is considerably weaker than Linear Logic: it has no axioms, the only rule is cut, and the only connective is linear negation.
3. Cerrito remarks that the completeness result would no longer be true in the presence of the contraction rule. (The reader should be aware that it is the absence of the classical structural rules weakening and contraction that primarily distinguish Linear Logic from from classical logic—in the absence of these rules, the classical connectives split into several alternatives.) However, if the format of the 'linear' theories $\mathbf{LT}_P$ is changed from one-sided sequents into two-sided sequents, contraction can be eliminated (cf. [GLT90].) In fact, it is only contraction with literals that correspond to the heads of clauses that is harmful.

4.5 Substructural Properties Revisited

Resuming some previous discussions on admissibility of (sub)structural rules, we remark the following (taking the literal formulation of the rules as given in Figure 3.1 and 3.2):
Obviously, Atomic Monotonicity is not an admissible rule for $\mathbf{S}_{P_r}$. An easy counterexample is the following: $[A \leftarrow B, D; A; D \leftarrow D] \Rightarrow A$ is derivable in $\mathbf{S}_{P_r}$, while $[A \leftarrow B, D; B; A; D \leftarrow D] \Rightarrow A$ is not derivable in $\mathbf{S}_{P_r}$. Also, it is not surprising that Rightward Extension and Rightward Contraction are admissible rules for $\mathbf{S}_{P_r}$, and that the classical structural rules Weakening, Leftward Contraction, and Classical Cut, are not admissible for $\mathbf{S}_{P_r}$.

However, while success of atoms is not preserved under exchange of clauses, finite failure of atoms is preserved under exchange:

4.5.1. LEMMA. *Let $\mathbf{S}_{P_r} \vdash [P] \Rightarrow \neg A$, and let R be a permutation of P. Then $\mathbf{S}_{P_r} \vdash [R] \Rightarrow \neg A$.*

PROOF. Let $\mathbf{S}_{P_r} \vdash [P] \Rightarrow \neg A$, and let, for some atom B, R be the result of a permutation π_b among the B-clauses of P. Let $\mathcal{T}$ be the (finitely failed) LD-tree for $P \cup \{A\}$. If B is never selected on $\mathcal{T}$, then $\mathcal{T}$ is also the finitely failed LD-tree for $R \cup \{A\}$. So suppose B is selected, say in nodes B, N_i. As $\mathcal{T}$ is finitely failed, all branches going through the nodes B, N_i end in a leaf marked as failed. Now assume that B is never selected in nodes below B, N_i. Applying the permutation π_b to the immediate descendents of all these nodes B, N_i results in the LD-tree for $R \cup \{A\}$. Otherwise, if B is selected below a node B, N_i, the permutation has to be executed stepwise, bottom up in $\mathcal{T}$. As all of the permuted branches are failed, the result is again a finitely failed LD-tree.

The lemma immediately follows. □

Also, finite failure (unlike success) of atoms is preserved under deletion of clauses. Let $R \prec P$ if R is the result of deleting clauses from P.

4.5.2. LEMMA. *Let $\mathbf{S}_{P_r} \vdash [P] \Rightarrow \neg A$, and let $R \prec P$. Then $\mathbf{S}_{P_r} \vdash [R] \Rightarrow \neg A$.*

PROOF. Let $\mathbf{S}_{P_r} \vdash [P] \Rightarrow \neg A$, and let $R \prec P$. Let $\mathcal{T}$ be the (finitely failed) LD-tree for $P \cup \{A\}$. The LD-tree for $R \cup \{A\}$ is obtained by removing those branches from $\mathcal{T}$ that correspond to deleted (occurrences of) clauses in P. In the resulting tree, there may be some unmarked leaves, for atoms B such that $R_B = \emptyset$. Mark these as failed. The result is the finitely failed LD-tree for $R \cup \{A\}$. □

Combining the above lemmas:

4.5.3. LEMMA. *Let $\mathbf{S}_{P_r} \vdash [P] \Rightarrow \neg A$, and let $R \subseteq P$. Then $\mathbf{S}_{P_r} \vdash [R] \Rightarrow \neg A$.*

Using the above lemma, we can show that both Full Context Cut and Partial Context Cut (cf. Figure 3.2) are admissible rules for $\mathbf{S}_{P_r}$. First, we prove a useful lemma.

4.5.4. LEMMA. *Let $\mathbf{S}_{P_r} \vdash [P; Q] \Rightarrow C$. Then*

1. $\mathbf{S}_{P_r} \vdash [P; C; Q] \Rightarrow C$, *2.* $\mathbf{S}_{P_r} \vdash [P; C; Q] \Rightarrow^* C$ *iff* $\mathbf{S}_{P_r} \vdash [P; Q] \Rightarrow^* C$.

PROOF. Suppose $[P; Q] \Rightarrow C$ is derivable in $\mathbf{S}_{P_r}$.
Then C cannot fail from $P; C; Q$, as that would contradict Lemma 4.5.2. Suppose that the search for C from $P; C; Q$ diverges. Then the search for C must diverge via one of the C-clauses in P. Again, this contradicts the assumption that $[P; Q] \Rightarrow C$ is derivable. Therefore, C succeeds from $P; C; Q$. This proves 1).
Let, by 1), $\mathcal{T}$ and $\mathcal{S}$ be the successful LD-trees for $P; C; Q \cup \{C\}$ respectively $P; Q \cup \{C\}$. It is immediate that $\mathcal{T}$ is finite if and only if $\mathcal{S}$ is finite: $\mathcal{S}$ can be obtained from $\mathcal{T}$ by deleting the (finite and successful) branch corresponding to the fact-clause C. This proves 2). □

Now we can prove that Full Context Cut is admissible for $\mathbf{S}_{P_r}$. In fact, we will prove admissibility of a stronger version of FCC, with, instead of just atomic consequents A, literal consequents L, i.e. consequents of the form A or $\neg A$.

4.5.5. LEMMA. *Full Context Cut is admissible for* $\mathbf{S}_{P_r}$.

PROOF. Suppose that C succeeds on $P; Q$, and that L succeeds on $P; C; Q$. The proof proceeds by induction on the weight of the seach tree $\mathcal{T}$ for $P; C; Q \cup \{L^+\}$.

The base case, $\omega(\mathcal{T}) = \langle 1, 1 \rangle$, is trivial: either $L = \top$ or L^+ is undefined in $P; C; Q$, and thus also undefined in $P; Q$.

[IH] Suppose the lemma holds for search trees with weight $<_o \langle k, m \rangle$.

Suppose that $\omega(\mathcal{T}) = \langle k, m \rangle$.
If $L^+ = C$, the lemma trivially holds, so we can assume without loss of generality, that $L^+ \neq C$.
Let N be a relevant descendant of the root of $\mathcal{T}$, that is, an immediate descendant that lies in $\mathcal{T}^s$. Consider the substree $\mathcal{S}$ of $\mathcal{T}$ generated by N. $\mathcal{S}$ is the search tree for $P; C; Q \cup \{N\}$; observe that $\omega(\mathcal{S}) <_o \omega(\mathcal{T})$. Let $\mathcal{S}$' be the search tree for $P; Q \cup \{N\}$. By the inductive hypothesis and by the lemmas 4.2.4 – 4.2.6, we have that
(*) $\mathcal{S}'$ is successful (successful and finite; failed) iff $\mathcal{S}$ is successful (successful and finite; failed).
The search path $\mathcal{V}^s$ of the search tree $\mathcal{V}$ for $P; Q \cup \{L^+\}$ can be constructed as follows:

Replace in $\mathcal{T}$ the subtrees $\mathcal{S}$ generated by the relevant descendants N of the root, by the search paths $\mathcal{S}'^s$ of the corresponding search trees $\mathcal{S}'$ for $P; Q \cup \{N\}$; in addition, remove all non-relevant descendants of the root. Observe that, because by assumption $L^+ \neq C$, the roots of $\mathcal{T}$ and $\mathcal{V}$ have the same relevant descendants.
By (*), $\mathcal{V}$ is successful (failed) if $\mathcal{T}$ is successful (failed). □

4.5.6. LEMMA. *Partial Context Cut is admissible for* $\mathbf{S}_{P_r}$.

PROOF. Suppose $\mathbf{S}_{P_r} \vdash [P] \Rightarrow C$, and $\mathbf{S}_{P_r} \vdash [Q; C; R; Z] \Rightarrow L$, where $P \subseteq Q; R$ and $R_C = \emptyset$.

Suppose that C fails from $Q; C; R; Z$. Then, by Lemma 4.5.3, it also fails from P, which contradicts our assumptions. So C succeeds from $Q; C; R; Z$.

We claim that C also succeeds from $Q; R; Z$. Let Z^- be the result of deleting all C-clauses from Z. If C fails from $Q; R; Z^-$, then, by Lemma 4.5.3, C would fail on

P, contradicting the assumption. If the search for C from $Q; R; Z^-$ diverges, then C would diverge via a C-clause in Q, as $R_A; Z_A^- = \emptyset$. But we have already established that C succeeds from $Q; C; R; Z$, and this excludes divergence via a C-clause in Q. Therefore, C succeeds on Q, R, Z.

Also, the (successful) LD-trees t and s for respectively $Q; C; R; Z \cup \{C\}$ and $Q; R; Z \cup \{C\}$ are either both finite or both infinite, as s can be obtained by deleting the finite successbranch corresponding to the fact clause $C \leftarrow$ from t.

The lemma now follows by the same line of reasoning used in the proof of the previous lemma 4.5.5. □

4.6 Notes

- The sections 4.1–4.5 are based on the ILLC research report CT-94-12 *Gentzen Systems for Logic Programming Styles*, Amsterdam, 1994 [Kal94], and will appear as the second part of [Kal95c] *Gentzen Systems for Logic Programming Styles II: Logics for Prolog* in the Bulletin of the IGPL, in 1995.
- The Prolog built-in `fail`, defined in Prolog as the automatically failing atom, can be defined in any of the above systems as the failure negation of the empty goal, that is, `fail` $:= \neg\top$.

Chapter 5

Prolog's cut

5.1 The Prolog Cut

An important non-declarative feature of Prolog is the cut operator ! , which allows for dynamic pruning of the search tree and control of the extensive Prolog backtracking mechanism. The Prolog cut has a non-declarative, dynamic flavour. It can change the declarative meaning of programs. In practice, the Prolog cut ! is used for various purposes. Efficiency of programs can be substantially improved by use of cut. The negation as failure rule for logic programs is implemented in Prolog by the cut ! — that is, it makes Prolog computation on normal programs possible. Further, various other built-in's are implemented by the use of cut.

The effect of the cut operator can be described as follows. Suppose a program P contains the clause $A \leftarrow B_1, \ldots, B_n, !, C_1, \ldots, C_m$ as part of the definition of A in P. Suppose that during a search for the goal A this clause is encountered. As soon as the cut operator is encountered as a goal, it succeeds and the search is committed to all choices made between the time the 'parent goal' A was encountered as a goal and the time this cut ! was activated. More specifically, as soon as ! is activated in the context of the above A-clause,

1. all other ways of resolving A via further A-clauses are discarded, and
2. all other ways of resolving any of the B_i are discarded.

It is important to observe that, while the cut prunes all the alternative solutions of the conjunction of the goals to its left, it does not affect the goals to its right.

The effect of the Prolog cut can be illustrated by the standard cut-fail definition of negation in Prolog:

$$\begin{array}{ll} (1) & \texttt{neg}(X) \leftarrow X, !, \texttt{fail} \\ (2) & \texttt{neg}(X) \leftarrow \end{array}$$

Observe that this definition employs Prolog's meta-variable facility: the variable X occurs as an atom in the body of the uppermost clause; as a result, the underlying syntax is ambivalent (cf. Chapter 1). Further, it employs the built-in `fail` that automatically fails.

A search for the goal $\mathtt{neg}(X)$ will first try to establish X, via clause (1). If X fails, the search backtracks via the last choice point ($\mathtt{neg}(X)$), proceeds via the second clause in the definition of $\mathtt{neg}(X)$, and succeeds. Otherwise, if X succeeds, the search proceeds via the goal !, which immediately succeeds and prevents eventual backtracking via X or $\mathtt{neg}(X)$; finally, `fail` immediately fails, and as all backtracking is prevented, $\mathtt{neg}(X)$ fails. Thus, $\mathtt{neg}(X)$ succeeds if X fails, while $\mathtt{neg}(X)$ fails if X succeeds.

While ideally, in Prolog programming, the relative order of the clauses in a program is irrelevant, in definitions involving cut, this order can be crucial. For example, if in the definition of the negation predicate above the order of the clauses were reversed, $\mathtt{neg}(X)$ would automatically succeed.

Another typical example of the use of cut in Prolog is the if-then-else control structure:

$$\begin{array}{l} if\_then\_else(P, Q, R) \leftarrow P, !, Q \\ if\_then\_else(P, Q, R) \leftarrow R \end{array}$$

The *if_then_else* predicate implements the usual "if P then Q, else R" construct of imperative programming languages. Declaratively, the relation holds if P and Q are true or if P is false and R is true ($(P \wedge Q) \vee (\neg P \wedge R)$). Procedurally, first a search for P is started. If P succeeds, a search for Q is started; if it fails, a search for R is started. Observe that $\mathtt{neg}(X)$ can be defined as $if\_then\_else(P, \mathtt{fail}, \top)$.

The procedural behaviour of the cut operator can (partially) be restated as a pruning operator on LD-trees, as follows. Let T be an LD-tree, and let Q be a node in T with ! as the leftmost (selected) atom. Let Q' be the origin of this cut, that is, the (unique) node in T that corresponds to its parent goal. Then execution of the cut atom involves pruning, from T, all branches that lie to the right of Q and contain Q'.

However, in the context of LD-trees, which are built up breadth-first, different cuts might interact. As an example, consider the following program, whose LD-tree is given in Figure 5.1.

$$\begin{array}{l} p \leftarrow q, !, t \\ p \leftarrow \\ q \leftarrow r, !, t \\ q \leftarrow \\ r \leftarrow \end{array}$$

The LD-tree corresponding to the program P contains two nodes with a cut atom as the selected atom, marked as a and b. Suppose they are processed from left to right. Then the cut marked as b prunes the middle branch of the tree, including the node with the cut marked as a. The resulting tree, consisting of the leftmost and the rightmost branch of the original tree, is successful. In contrast, suppose that the cuts are processed from right to left. Then the cut marked as a is processed first and prunes the rightmost branch of the tree. The cut marked as b in its turn then prunes the middle branch of the original tree, and the resulting tree is failed.

Under Prolog's depth-first strategy, the cuts are processed from left to right, and the cut marked as a would be processed first. The above example shows that

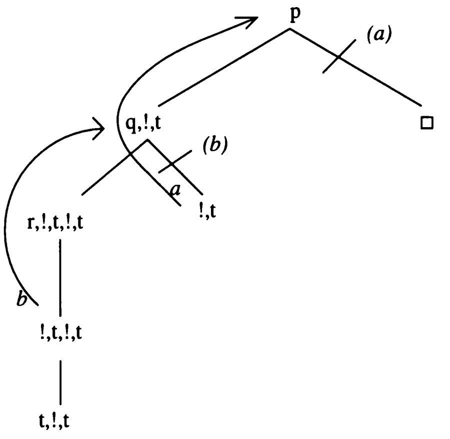


Figure 5.1: cut-order in LD-trees

LD-trees are not the most efficient structures to account for Prolog's computation mechanism if the effect of cut is to be reflected properly. In Apt & Teusink [AT95] the concept of P-trees was introduced as an elegant alternative to account for Prolog's computation mechanism incorporating cut. Unlike LD-trees, P-trees are built up depth-first. In each stage of their construction, the only leaves that are generated are the direct successors of the leftmost unmarked leaf. The involved Prolog backtracking mechanism is automatically accounted for, and cut is incorporated in a natural way as a simple pruning operator.

Due to the procedural nature of the Gentzen calculus $\mathbf{S}_{Pr}$ for Prolog, this calculus, or rather its version $\mathbf{S}^-_{Pr}$, provides a natural setting for the incorporation of the Prolog cut. In the remainder of the present chapter we will discuss a modification of $\mathbf{S}^-_{Pr}$ which accounts for Prolog computation on programs with cut. This new calculus, $\mathbf{S}_{Pr!}$, consists of a slight modification of the $\mathbf{S}^-_{Pr}$ rules, plus two extra rules, reflecting the pruning due to activated cuts. Correctness of the calculus $\mathbf{S}_{Pr!}$ will be proved using Apt & Teusink's P-trees.

The Prolog cut operator consists in fact of two components, each corresponding to a distinct pruning operator. The *soft cut* component causes pruning of all further clauses for the origin of the cut. The *one solution* component prunes further alternative ways of solving subgoals before the activated cut. The usual Prolog cut operator can be interpreted as the composition of these two operators. We will show that both the soft cut and the one solution cut can be incorporated in the Gentzen format, by appropriate and natural modifications of the cut-specific rules of $\mathbf{S}_{Pr!}$.

Notation and conventions

- The propositional language of the previous chapters is extended with the cut atom !.
- *(Prolog) atoms* are either pure atoms $A, B, C, \ldots$, or $\top$, or !.

- $\mathcal{A}$ is a *(Prolog) clause* if $\mathcal{A}$ is $A \leftarrow B_1, \ldots, B_n$, where A is a pure atom and the B_i are (Prolog) atoms.
- $\mathcal{A}$ is a *pure (Prolog) clause* if $\mathcal{A}$ is $A \leftarrow B_1, \ldots, B_n$, where all of the B_i are pure atoms or $\top$.
- P is a *(Prolog) program* if P is a finite list of (Prolog) clauses.
- P is a *pure (Prolog) program* if all the clauses in P are pure Prolog clauses.
- To save space in the notation of rules (and derivations), we use the 'sum' $\sum_{i=1}^{n} [P_i] \Rightarrow E_i$ as a shorthand for the the list of n sequents $[P_1] \Rightarrow E_1$, ..., $[P_n] \Rightarrow E_n$.
- We will distinguish *queries* and *pure (Prolog) queries.* We will use bod face capitals $\mathbf{B}, \mathbf{C}, \mathbf{E}$ to indicate pure Prolog queries, that is, finite lists $B_1, \ldots, B_n$ etc. of atoms in which the cut ! does not occur. In contrast, queries in which ! possibly occur, will be indicated by a capital Q (or Q', Q_1, etc.) Thus we do not use Q anymore to indicate programs.

Observe that, by the above conventions, clauses never have ! as a head. We need an additional definition for queries in which cuts occur.

5.1.1. Definition. For a (possibly empty) list of atoms $\mathbf{B}$, $(\mathbf{B} \lhd !)$ indicates the set of lists that are 'paddings' of $\mathbf{B}$ with the Prolog cut ! . That is, the set $(\mathbf{B} \lhd !)$ is defined as follows:

1. $\mathbf{B} \in (\mathbf{B} \lhd !)$
2. If Q_1 and Q_2 in $(\mathbf{B} \lhd !)$, then also $Q_1, !, Q_2 \in (\mathbf{B} \lhd !)$.

We will use the notation $(\mathbf{B} \lhd !)$ to indicate any element of the padding of $\mathbf{B}$.

5.2 Axiomatising cut

The Prolog cut can be incorporated in the calculus $\mathbf{S}^-_{P_r}$, discussed in the previous sections, as follows.

1. The (notation of) the original rules of $\mathbf{S}^-_{P_r}$ is slightly modified, to incorporate possible cuts in bodies of clauses. Also,
2. Two rules (respectively a new version of a modus ponens and a prefixing rule) are added, accounting for the specific pruning caused by the Prolog cut.

The result is a calculus $\mathbf{S}_{P_r!}$, given in the Figures 5.2 and 5.3, which is correct for Prolog computation on Prolog programs. The present section is devoted to a description and motivation of the various rules of $\mathbf{S}_{P_r!}$.

$\mathbf{S}_{P_r!}$ inherits, without any alterations, the axioms of $\mathbf{S}^-_{P_r}$, expressing immediate success of the empty goal and immediate failure of undefined atoms. It is useful to observe here that there is no separate $\mathbf{S}_{P_r!}$ rule expressing the immediate success of the cut.

$\mathbf{S}_{P_r!}$ inherits the rules for heterogeneous permutation HP and HP* from $\mathbf{S}^-_{P_r}$: prunings due to the activation of a cut are independent of the relative order of clauses with different heads.

The rules MP# 1 and MP# 2 are respectively the $\mathbf{S}^-_{P_r}$-rules MP# and MP*. Further, the rules $\text{PFX}_{1!}$, $\text{PFX}_{2!}$, and $\text{PFX}_{3!}$ are immediate generalisations of the $\mathbf{S}^-_{P_r}$-rules PFX_1, PFX_2, and PFX_3: the failing body atom D prevents a computation for any of the body atoms to its right, so there may be cut atoms in these positions.

The various effects of the activation of a cut atom are in fact only reflected in the three rules $\text{MP}_!$, $\text{MP}^*_!$, and $\text{PFX}_{\neg 2!}$.

- $\text{MP}_!$: ! succeeds immediately
  Suppose $B_1, \ldots, B_n$ succeed from $A \leftarrow A; P$. Then the $B_1, \ldots, B_n$ succeed from P and in the respective search-trees, A is never encountered as a goal. Now prefix P with a clause $A \leftarrow Q$, where $Q \in (\mathbf{B} \lhd !)$, and start a search for A from the new program, via the added clause. The B_i all succeed from the new program. Also, the cut atoms immediately succeed. Therefore, the composite goal Q succeeds. So the search for A succeeds via the leftmost A-clause $A \leftarrow Q$.
- $\text{MP}^*_!$: ! prunes remaining alternative proofs for body atoms preceding it, and also prunes remaining alternative proofs of its origin.
  Suppose $B_1, \ldots, B_n$ succeed from $A \leftarrow A; P$; also suppose that $C_1, \ldots, C_m$ succeed from $A \leftarrow A; P$, and that the respective search-trees are finite. Now prefix the program P with a clause $A \leftarrow Q$, where $Q \in (\mathbf{B} \lhd !)$, and start a search for A from the new program, that is, via the new A-clause. As in the above case, the B_i succeed on the new program, by the assumptions, and the cut atoms possibly present in Q succeed immediately. Finally, the last (explicit) cut atom is reached. It succeeds, it deletes remaining alternative proofs for the B_i, and it prunes all further A-clauses in P from the search tree. The search then continues with a search for the C_i in the context of the new program. By the assumptions, the C_i all succeed, and their search tree is finite. Thus A succeeds, and, due to the pruning, the complete search tree is finite.
- $\text{PFX}_{\neg 2!}$: ! prunes alternative proofs for body atoms preceding it, and also prunes alternative proofs of its origin.
  Suppose $C_1, \ldots, C_n$ succeed from $A \leftarrow A; P$. Suppose $B_1, \ldots, B_m$ succeed from $A \leftarrow A; P$, and suppose that the respective search-trees for the B_i are finite. Also suppose that D fails finitely from $A \leftarrow A; P$. Now prefix the program A with a new clause $A \leftarrow Q_1, !, \mathbf{B}, D, Q_2$. where where $Q_1 \in (\mathbf{C} \lhd !)$, and where $Q_2 \in (\mathbf{E} \lhd !)$. Start a search for A in the context of the new program, via the goal $Q_1, !, \mathbf{B}, D, Q_2$. Again, as in the above case, $Q_1, !, \mathbf{B}$ succeeds, and its search tree is finite. Then a search for D is started, which fails finitely. Backtracking via the alternative clauses for the B_i will eventually result in (finite) failure, due to the assumption that the search trees for the B_i are finite. There is no backtracking via the C_i, because of the pruning due to the cut atom preceding $\mathbf{B}$. Finally, there is no backtracking via the root A of the tree: as A is the origin of the cut atom preceding $\mathbf{B}$, alternative clauses for A (in P) cannot be entered upon backtracking via the top of the search tree A. Thus the search for A in the context of the new program fails finitely.

This completes the description of the rules of $\mathbf{S}_{P_r!}$.

We can, in the set of rules of $\mathbf{S}_{P_r!}$, distinguish the following cases:

- *success* (AX and AX$*$, the modus ponens rules, $\mathrm{PFX}_{1!}$, $\mathrm{PFX}_{3!}$) versus *failure* ($\mathrm{AX}_{\neg}$, $\mathrm{PFX}_{2!}$, and $\mathrm{PFX}_{\neg 2!}$)
- *(possibly) infinite success* ($\mathrm{MP}_!$ and $\mathrm{PFX}_{1!}$) versus *finite* success (the remaining modus ponens rules, and $\mathrm{PFX}_{3!}$)
- success via *leftmost* clause (all of the modus ponens rules) versus success via some *further* clause ($\mathrm{PFX}_{1!}$ and $\mathrm{PFX}_{3!}$).
- *finiteness* due to an activated *cut* ($\mathrm{MP}_!^*$and $\mathrm{PFX}_{\neg 2!}$) versus *finiteness not due to an activated cut* (MP# 1, MP# 2, $\mathrm{PFX}_{2!}$, and $\mathrm{PFX}_{3!}$).

We will show that $\mathbf{S}_{Pr!}$ is correct with respect to Prolog computation, that is, we will prove the following theorem:

5.2.1. THEOREM.

1. *A succeeds via Prolog computation iff* $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow A$*;*
2. *A finitely fails under Prolog computation iff* $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow \neg A$.

The next few sections will be devoted to a proof of the above correctness theorem.

5.2.2. EXAMPLE. We give an example of a derivation in the calculus $\mathbf{S}_{Pr!}$. Consider the program P:

$$\begin{array}{l} A \leftarrow B, !, D \\ A \leftarrow C \\ B \leftarrow \\ C \leftarrow \end{array}$$

A finitely fails from P. The derivation of the corresponding sequent, omitting obvious subderivations, is the following:

$$\frac{[A \leftarrow A; A \leftarrow C; B; C] \Rightarrow B \quad [A \leftarrow A; A \leftarrow C; B; C] \Rightarrow \neg D}{[A \leftarrow B, !, D; A \leftarrow C; B; C] \Rightarrow \neg A}\ \mathrm{PFX}_{\neg 2!}$$

□

5.3 P-trees

We will prove the correctness of the calculus $\mathbf{S}_{Pr!}$ with respect to Prolog computation using the concept of P-*trees*, introduced by Apt and Teusink [AT95]. As argued in Section 5.1, P-trees are better suitable to model the computation process of Prolog, including cut, than LD-trees. In our discussion of P-trees in the present section, we follow the exposition of [AT95].

P-trees are defined as the limit of a sequence of *pre*-P-*trees*. Before we can introduce the notion of pre-P-tree, we need the somewhat more liberal notion of *semi*-P-*tree*.

5.3.1. DEFINITION. A *semi*-P-*tree* is an ordered tree whose nodes contain queries and whose leaves are possibly marked as *successful* or *failed.*

$\mathbf{S}_{Pr!}$

Axioms

$[P] \Rightarrow \top$ (AX)

$[P] \Rightarrow^{*} \top$ (AX*)

$[P] \Rightarrow \neg A$ if $P_A = \emptyset$ (AX$_\neg$)

HP/HP$*$

$$\frac{[P;\mathcal{D};\mathcal{C};R] \Rightarrow L}{[P;\mathcal{C};\mathcal{D};R] \Rightarrow L}\text{HP} \qquad \frac{[P;\mathcal{D};\mathcal{C};R] \Rightarrow^{*} L}{[P;\mathcal{C};\mathcal{D};R] \Rightarrow^{*} L}\text{HP}^{*}$$

if $\mathcal{D}$ and $\mathcal{C}$ are two clauses with different heads.

MP$_!$

$$\frac{\sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow B_i}{[A \leftarrow (\mathbf{B} \triangleleft !); P] \Rightarrow A}$$

MP# 1

$$\frac{[P] \Rightarrow^{*} A \quad \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^{*} B_i}{[A \leftarrow \mathbf{B}; P] \Rightarrow^{*} A}$$

MP# 2

$$\frac{[P] \Rightarrow \neg A \quad \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^{*} B_i}{[A \leftarrow \mathbf{B}; P] \Rightarrow^{*} A}$$

MP$^{*}_{!}$

$$\frac{\sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow B_i \quad \sum_{i=1}^{m} [A \leftarrow A; P] \Rightarrow^{*} C_i}{[A \leftarrow (\mathbf{B} \triangleleft !), !, \mathbf{C}; P] \Rightarrow^{*} A}$$

Figure 5.2: Axioms, Heterogeneous Permutation, Modus Ponens of $\mathbf{S}_{Pr!}$

$\text{PFX}_{1!}$

$$\frac{[P] \Rightarrow A \qquad \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow A}$$

$\text{PFX}_{2!}$

$$\frac{[P] \Rightarrow \neg A \qquad \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

$\text{PFX}_{\neg 2!}$

$$\frac{\sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow C_i \quad \sum_{i=1}^{m} [A \leftarrow A; P] \Rightarrow^* B_i \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

$\text{PFX}_{3!}$

$$\frac{[P] \Rightarrow^* A \qquad \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow^* A}$$

Figure 5.3: Prefixing rules of $\mathbf{S}_{Pr!}$

The above notion of semi-P-tree is slightly different from the notion defined in [AT95]: as we deal here with propositional programs instead of predicate programs, computations can in our case not end in error — therefore, we only need the markings *successful* and *failed.*

In order to define the pruning behaviour of the cut operator on semi-P-trees, we need the fact that, in ordered trees, there is a strict order among sibling nodes (nodes with the same parent).

5.3.2. DEFINITION. *Let n and m be two nodes in an ordered tree.* n lies to the right of m *if there are predecessors n' and m' of n and m respectively, such that n' and m' are siblings and m' is strictly smaller (in the tree order) than n'.*

Further, we need the notion of the origin of a cut atom in the context of semi-P-trees.

5.3.3. DEFINITION. Let $b = Q_1, \ldots, Q_n$ be a branch in a semi-P-tree, and let Q_k be a node in b with ! as the leftmost atom. The *origin* of this occurrence of ! is the first predecessor of Q_k containing less atoms ! than Q_k, if there is such a predecessor; otherwise the origin of this occurrence of cut is the root Q_1 of the tree.

We are now able to define the following pruning operator on trees, which effectuates the pruning due to the Prolog cut !.

5.3.4. DEFINITION. Let $\mathcal{T}$ be a semi-P-tree, let Q be a node in $\mathcal{T}$ with ! as the leftmost atom, and let Q' be the origin of this occurrence of ! . The operator $cut(\mathcal{T}, Q, Q')$ removes all nodes from $\mathcal{T}$ that are descendants of Q' and lie to the right of Q.

We will be only interested in semi-P-trees that correspond to initial stages of a Prolog computation with respect to a fixed (atomic Prolog) program. In fact, each stage in such a computation corresponds to exactly one semi-P-tree. Of course, not every semi-P-tree needs to correspond to a Prolog computation. Semi-P-trees that do correspond to Prolog computations on a fixed program, are *pre-*P*-trees.* These are defined inductively, using the following extension operator on semi-P-trees. In fact, the application of this extension operator corresponds to doing one step in the usual Prolog computation process.

5.3.5. DEFINITION. Let $\mathcal{T}$ be a semi-P-tree and let P be a Prolog program. Let Q be the leftmost unmarked leaf of $\mathcal{T}$.
The *extension* $ext_P(\mathcal{T})$ of T w.r.t. P is obtained as follows:

1. If Q is the empty query or a list in which every element is $\top$, mark Q as *successful.*
2. Otherwise, let Q be of the form A, Q_1.
   (a) If A is an undefined pure atom in the program P, mark Q as *failed.*
   (b) Otherwise, if A is a pure atom, add for every clause $A \leftarrow B_1, \ldots, B_k$ in P, a node $B_1, \ldots, B_k, Q_1$ as a child of Q. Order these new nodes according to the order of the A-clauses in P.
   (c) Otherwise, if A is !, let Q' be the origin of this occurrence of ! in Q. Apply the operation $cut(\mathcal{T}, Q, Q')$; then add, in the resulting tree, Q_1 as a single child to the node Q.

(d) If $\mathcal{T}$ does not have unmarked leaves, then $ext_P(\mathcal{T}) = \mathcal{T}$.

Observe that, given a semi-P-tree $\mathcal{T}$ and a propositional Prolog program P, $ext_P(\mathcal{T})$ is unique.

We are now in position to define the class of pre-P-trees.

5.3.6. Definition. Let P be a Prolog program. The class of *pre-P-trees w.r.t. P* consists of the following semi-P-trees:

- For every query Q, the tree consisting of the single node Q is a pre-P-tree w.r.t. P. In particular, these trees are *initial* pre-P-trees w.r.t. P.
- If $\mathcal{T}$ is a pre-P-tree w.r.t. P, then $ext_P(\mathcal{T})$ is a pre-P-tree w.r.t. P.

P-trees, which model 'completed' Prolog computations on Prolog programs, are limits of possibly infinite sequences of pre-P-trees. The usual notion of tree-extension is not appropriate to define the limit involved in their definition: because of the pruning due to cut, $ext_P(\mathcal{T})$ is not necessarily a tree extension of $\mathcal{T}$ in the usual sense. We use the following notion of inclusion between semi-P-trees, due to [AT95].

5.3.7. Definition. Let $\mathcal{T}$ and $\mathcal{T}'$ be semi-P-trees.

1. $\mathcal{T} \sqsubseteq \mathcal{T}'$ if $\mathcal{T}'$ is the result of
   (a) adding a finite number of children to a leaf of $\mathcal{T}$, or
   (b) removing a single subtree from $\mathcal{T}$ provided the root of this subtree is not an only child.
2. $\mathcal{T} \sqsubset \mathcal{T}'$ if $\mathcal{T} \sqsubseteq \mathcal{T}'$ and $\mathcal{T}' \not\sqsubseteq \mathcal{T}$.
3. $\subseteq$ is the reflexive and transitive closure of $\sqsubset$. □

The following result is proven in [AT95].

5.3.8. Lemma. *$\subseteq$ is a partial order on the class of semi-P-trees.*

Observe that the extensions follow the partial order:

5.3.9. Lemma. *Let $\mathcal{T}$ be a pre-P-tree for Q w.r.t. P. Then $\mathcal{T} \subseteq ext_P(\mathcal{T})$.*

By Lemma 5.3.8, the following notion of P-tree is well-defined.

5.3.10. Definition. A *P-tree* for $P \cup \{Q\}$, where P is a Prolog program and Q is a Prolog query, is the limit (in the sense of $\subseteq$) of a sequence $\mathcal{T}_0, \mathcal{T}_1, \ldots$, such that $\mathcal{T}_0$ is an initial P-tree consisting of the query Q, and for all i, $\mathcal{T}_{i+1} = ext_P(\mathcal{T}_i)$.
In that case, $\mathcal{T}_0, \mathcal{T}_1, \ldots$ is the *sequence of pre-P-trees associated to $\mathcal{T}$*. □

Note that a P-tree can have unmarked leaves. In that case, the P-tree will have exactly one infinite branch to the left of all unmarked leaves. These unmarked leaves then correspond to resolvents the Prolog computation did not reach because it got trapped in the infinite derivation corresponding to the infinite branch.

5.3.11. Definition.
A P-tree for $P \cup \{Q\}$ is *successful* if it contains a branch ending in a leaf marked as successful.
A P-tree for $P \cup \{Q\}$ is *finitely failed* if all its branches end in a leaf marked as failed.
□

P-trees and Prolog computations are related, as expected, as follows:

5.3.12. LEMMA. *Let P be a Prolog program, and Q a query.*

1. *Q succeeds via Prolog computation iff the* P*-tree for $P \cup \{Q\}$ is successful;*
2. *Q fails finitely via Prolog computation iff the* P*-tree for $P \cup \{Q\}$ is finitely failed.*

The above lemma plays a crucial role in the coming sections, where we set out to prove the correctness of $\mathbf{S}_{Pr!}$ with respect to Prolog computation, Theorem 5.2.1. The above lemma enables us to reason on P-trees. It should be observed that in general, P-trees and LD-trees do not coincide. However, in some special cases, the P-tree is identical to the LD-tree.

5.3.13. PROPOSITION. *Let $\mathcal{T}$ be the finite* P*-tree for $P \cup \{Q\}$, where Q is a pure Prolog query, and P is a pure Prolog program. Then $\mathcal{T}$ is the LD-tree for $P \cup \{Q\}$. In particular, $\mathcal{T}$ is successful (finitely failed) as a* P*-tree iff $\mathcal{T}$ is successful (finitely failed) as an LD-tree.*

5.4 Some useful properties of P-trees

The present section is devoted to some preliminary results on P-trees, which we will use the Sections 5.5 and 5.7, where we prove soundness, respectively completeness of $\mathbf{S}_{Pr!}$. First we define some useful concepts.

5.4.1. DEFINITION. Let $\mathcal{T}$ be a successful P-tree for $P \cup \{Q\}$. The *successbranch* of $\mathcal{T}$ is the unique branch $b_{\mathcal{T}}$ such that
1) $b_{\mathcal{T}}$ ends in a leaf marked as successful
2) all branches left of $b_{\mathcal{T}}$ end in a leaf marked as failure.

5.4.2. DEFINITION. Let $\mathcal{T}$ be the successful P-tree for $P \cup \{Q\}$. Then $\mathcal{T}^s$ is the (unique) pre-P-tree in the associated sequence of pre-P-trees such that $\mathcal{T}^s$ contains $b_{\mathcal{T}}$ and all branches left of the successbranch $b_{\mathcal{T}}$, while the leaf of $b_{\mathcal{T}}$ is unmarked.

5.4.3. DEFINITION. Let N be a node in a pre-P-tree or a P-tree $\mathcal{T}$. A is *selected in* the node N if $N = A, M$ for some M. A is *selected on* the tree $\mathcal{T}$ if $\mathcal{T}$ contains a node N in which A is selected, such that N does not lie to the right of the leftmost unmarked leaf of $\mathcal{T}$.

Observe that, according to the above definition, it is not necessarily true that, if A is not selected on a tree $\mathcal{T}$, there are no nodes in $\mathcal{T}$ in which A is not selected. As an example, let $\mathcal{T}$ be the P-tree for $A \leftarrow B; B \leftarrow; B \leftarrow B; A \leftarrow A \cup \{A\}$. A is not selected on $\mathcal{T}$ except in the root, but A is selected in the rightmost node of $\mathcal{T}$. We need this convention for an easy formulation of propositions like 5.4.6.

The following propositions express the proper equivalents of the earlier occurrence, transformation, and inessential difference properties in the context of P-trees.

5.4.4. PROPOSITION. *Let $\mathcal{T}$ be the successful* P*-tree for $P \cup \{A\}$.*

1. *If $\mathcal{T}$ is successful, then A is never selected on $\mathcal{T}^s$, except in the root of $\mathcal{T}$.*

2. *If $\mathcal{T}$ is finite, then A is never selected on $\mathcal{T}$, except in the root of $\mathcal{T}$.*

5.4.5. PROPOSITION. *Let $\mathcal{T}$ be the P-tree for $A \leftarrow Q; P \cup \{Q\}$. Then*

1. *If $\mathcal{T}$ is successful, A is never selected on $\mathcal{T}^s$.*
2. *If $\mathcal{T}$ is finite, then A is never selected on $\mathcal{T}$.*

5.4.6. PROPOSITION. *Let $\mathcal{T}$ be a successful P-tree for $P \cup \{Q\}$, and let A be a pure atom such that A is never selected on $\mathcal{T}^s$. Let R be a Prolog program such that P and R differ only w.r.t. A-clauses. Then*

1. *$\mathcal{T}^s$ is a pre-P-tree for $R \cup \{Q\}$, and*
2. *the P-tree for $R \cup \{Q\}$ is successful.*
3. *If A is never selected on $\mathcal{T}$, then $\mathcal{T}$ is the P-tree for $R \cup \{Q\}$.*

5.4.7. PROPOSITION. *Let $\mathcal{T}$ be a finitely failed P-tree for $P \cup \{Q\}$, and let A be a pure atom such that A is never selected on $\mathcal{T}$. Let R be a Prolog program such that P and R differ only w.r.t. A-clauses. Then $\mathcal{T}$ is also the finitely failed P-tree for $R \cup \{Q\}$.*

5.4.8. PROPOSITION. *Let $\mathcal{T}$ be the P-tree for $A \leftarrow A; P \cup \{Q\}$. Then*

1. *If $\mathcal{T}$ is successful then A is never selected on $\mathcal{T}^s$.*
2. *If $\mathcal{T}$ is finite, then A is never selected on $\mathcal{T}$.*

5.4.9. PROPOSITION. *Let $\mathcal{T}$ be the P-tree for $A \leftarrow A; P \cup \{Q\}$. Let $\mathcal{S}$ be the* P*-tree for $A \leftarrow Q; P \cup \{Q\}$. Then*

1. *If $\mathcal{T}$ is successful, then $\mathcal{S}$ is successful, and A is never selected on $\mathcal{S}^s$.*
2. *If $\mathcal{T}$ is finite, then $\mathcal{S} = \mathcal{T}$; in particular, if $\mathcal{T}$ is finitely failed, $\mathcal{S}$ is.*

PROOF. This follows immediately from the above propositions 5.4.6 – 5.4.8. □

5.4.10. DEFINITION. Let $\mathcal{T}$ be a semi-P-tree, and let A be an atom. $A \uplus \mathcal{T}$, the *A-extension of $\mathcal{T}$*, denotes the semi-P-tree with root A and $\mathcal{T}$ as the subtree generated by the only child of this root.

5.4.11. PROPOSITION. *Let $\mathcal{T}$ be the successful* P*-tree for $A \leftarrow Q; P \cup \{Q\}$. Then the* P*-tree for $A \leftarrow Q; P \cup \{A\}$ is successful.*

PROOF. By Proposition 5.4.5, A is never selected on $\mathcal{T}^s$. By definition, the extension step for $\mathcal{T}^s$ w.r.t. $A \leftarrow Q; P$ consists in marking the leftmost unmarked leaf as successful. Now consider $A \uplus \mathcal{T}^s$. A pre-P-tree $\mathcal{S}$ for $A \leftarrow Q; P \cup \{A\}$ is obtained by adding appropriate single children (corresponding to the bodies of the A-clauses in P) to the root A of $\mathcal{S}$. Also, the extension step for $\mathcal{S}$ w.r.t. $A \leftarrow Q; P$ consists in marking the leftmost unmarked leaf of the subtree $\mathcal{T}^s$ as successful. □

5.4.12. PROPOSITION. *Let $\mathcal{T}$ be the finite* P*-tree for $P \cup \{Q\}$, where Q is a pure Prolog query. Let A be an atom such that A is never selected on $\mathcal{T}$, and let S be the* P*-tree for $P \cup \{A\}$. Then*

1. *The* P*-tree $\mathcal{V}$ for $A \leftarrow Q; P \cup \{A\}$ is obtained by adding, to every node in $\mathcal{S}$ in which A is selected, a new leftmost child that generates T as a subtree.*

2. *$\mathcal{V}$ is successful if*
   (a) *$\mathcal{T}$ is successful or*
   (b) *$\mathcal{T}$ is finitely failed and $\mathcal{S}$ is successful.*
3. *$\mathcal{V}$ is finite if $\mathcal{S}$ is finite.*
4. *$\mathcal{V}$ is finitely failed if both $\mathcal{T}$ and $\mathcal{S}$ are finitely failed.*

5.4.13. DEFINITION. Let $\mathcal{T}$ be a semi-P-tree, and let R be a Prolog query. The *sum* $\mathcal{T} \oplus R$ is the result of replacing all nodes N of $\mathcal{T}$ by nodes N, R.

5.4.14. PROPOSITION. *Let Q be a Prolog query, let R be a pure Prolog query, and let $\mathcal{T}$ and $\mathcal{S}$ be successful* P*-trees for respectively $P \cup \{Q\}$ and $P \cup \{R\}$. Then the* P*-tree for $P \cup \{Q, R\}$ is successful.*

PROOF. Let $\mathcal{V}$ be the P-tree for $P \cup \{Q, R\}$. Replace the leftmost unmarked leaf in $\mathcal{T}^s \oplus R$ by $\mathcal{S}^s$. The resulting tree $\mathcal{V}^s$. □

5.4.15. PROPOSITION. *Let $\mathcal{T}$ be the successful* P*-tree for $P \cup \{Q\}$. Let R be a pure Prolog query, and let $\mathcal{S}$ be the finite* P*-tree $P \cup \{R\}$. Let $\mathcal{V}$ be the* P*-tree for $P \cup \{Q, R\}$. Then*

1. *$\mathcal{V}$ is successful if $\mathcal{S}$ is successful*
2. *$\mathcal{V}$ is finitely failed if $\mathcal{S}$ is finitely failed and $\mathcal{T}$ is finite*
3. *$\mathcal{V}$ is finite if both $\mathcal{T}$ and $\mathcal{S}$ are finite*

PROOF. Replace every node N in $\mathcal{T}$ by N, R. In the resulting semi-P-tree $\mathcal{T} \oplus R$, replace every leaf marked as successful by the tree $\mathcal{S}$. The resulting semi-P-tree, $\mathcal{V}$, is the P-tree for $P \cup \{Q, R\}$. □

Observe that in the above Propositions 5.4.14 and 5.4.15 it is crucial that R is a pure Prolog query, i.e., R does not contain the Prolog cut !. Also, the construction in the proof of Proposition 5.4.15 is allowed because $\mathcal{S}$ is finite.

5.4.16. PROPOSITION. *Let $\mathcal{T}$ be a successful* P*-tree for $P \cup \{Q\}$.
Then $ext(ext(\mathcal{T}^s \oplus !))$ is the finite and successful* P*-tree for $P \cup \{Q, !\}$.*

5.4.17. PROPOSITION. *Let $B_1, \ldots, B_n$ be pure atoms, and let, for $i \in [1, n]$, $\mathcal{T}_i$ be the successful* P*-tree for $P \cup \{B_i\}$. Let $Q \in (B_1, \ldots, B_n) \lhd !$, and let $\mathcal{T}$ be the* P*-tree for $P \cup \{Q\}$. Then*

1. *$\mathcal{T}$ is successful.*
2. *If A is never selected on any of the $\mathcal{T}_i^s$, then A is never selected on $\mathcal{T}^s$.*
3. *If the $\mathcal{T}_i$ are all finite then (a) $\mathcal{T}$ is finite, and (b) if A is not selected on any of the $\mathcal{T}_i$, then A is not selected on $\mathcal{T}$.*

PROOF. The lemma is an immediate consequence of the above propositions 5.4.15 and 5.4.16. □

5.4.18. PROPOSITION. *Let $\mathcal{T}$ be the finitely failed* P*-tree for $P \cup \{Q\}$, and let R be a query. Then the* P*-tree for $P \cup \{Q, R\}$ is finitely failed.*

PROOF. $\mathcal{T} \oplus R$ is the finitely failed P-tree for $P \cup \{Q, R\}$. □

5.4.19. PROPOSITION. *Let $\mathcal{T}$ be the successful* P*-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{Q_1, !, Q_2\}$. Then* P*-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$ is successful, and it is finite if /calT is finite.*

PROOF. Let $\mathcal{S}$ be the semi-P-tree with root A and with $ext(\mathcal{T}^s)$ as the subtree generated by the only descendant of the root A. Observe that A is never selected in $\mathcal{T}^s$. Thus $\mathcal{S}$ is a pre-P-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$, and witnesses the success of the P-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$.
In case $\mathcal{T}$ is also finite, A is never selected on $\mathcal{T}$. Let $\mathcal{V}$ be the tree with root A and $\mathcal{T}$ as the subtree generated by the sole immediate descendant of this root. Then $\mathcal{V}$ is the successful and finite P-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$. □

5.4.20. PROPOSITION. *Let the* P*-tree for $A \leftarrow A; P \cup \{Q_1\}$ be successful. Let the* P*-tree for $A \leftarrow A; P \cup \{Q_2\}$ be finitely failed. Then the* P*-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$ is finitely failed.*

PROOF. Let $\mathcal{S}$ be the P-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{Q_1, !, Q_2\}$. Under the conditions of the proposition, $\mathcal{S}$ is finitely failed, by the propositions 5.4.8, 5.4.16, and 5.4.15(2). Let $\mathcal{T}$ be the pre-P-tree with root A and $\mathcal{S}$ as the subtree generated by the unique child of the root A. Then $\mathcal{T}$ is the finitely failed P-tree for $A \leftarrow Q_1, !, Q_2; P \cup \{A\}$. □

5.5 Soundness

Using the results of the previous section, we can now prove soundness of the $\mathbf{S}_{Pr!}$ rules.

5.5.1. LEMMA.

1. *If $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow A$ then the* P*-tree for $P \cup \{A\}$ is successful.*
2. *If $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow^* A$ then the* P*-tree for $P \cup \{A\}$ successful and finite.*
3. *If $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow \neg A$ then the* P*-tree for $P \cup \{A\}$ finitely failed.*

PROOF. By induction on the length of derivations in $\mathbf{S}_{Pr!}$. It is trivial to check that the lemma hold for derivations of depth 1, that is, derivations consisting of the axioms.
INDUCTIVE HYPOTHESIS Suppose the lemma holds for sequents with $\mathbf{S}_{Pr!}$-derivations of depth $< n$.

Suppose δ is a derivation of depth n. We distinguish the following cases, according to the last rule applied in δ.

- Suppose that the last rule applied in δ is HP, and that δ is of the following form.

$$\frac{\begin{array}{c}\vdots\\ [P; \mathcal{D}, \mathcal{C}; R] \Rightarrow A\end{array}}{[P; \mathcal{C}, \mathcal{D}; R] \Rightarrow A.}$$

  By the Inductive Hypothesis we can assume that there exists a successful P-tree, say $\mathcal{T}$, for $P; \mathcal{D}, \mathcal{C}; R \cup \{A\}$. From the definition of P-trees it follows that P-trees are insensitive to the relative order of clauses with different heads. Thus $\mathcal{T}$ is also the successful P-tree for $P; \mathcal{C}, \mathcal{D}; R \cup \{A\}$.

The other cases are similar.

- The last rule applied is MP$_!$, so δ is of the following form:

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ [A \leftarrow A; P] \Rightarrow B_1 & \ldots & [A \leftarrow A; P] \Rightarrow B_n\end{array}}{[A \leftarrow (\mathbf{B} \lhd !); P] \Rightarrow A}$$

Let, by the Inductive Hypothesis, for $i \in [1, n]$, $\mathcal{T}_i$ be successful P-trees for $A \leftarrow A; P \cup \{B_i\}$, and let $Q \in (B_1, \ldots, B_n) \lhd !$. By Proposition 5.4.17(1), the P-tree $\mathcal{T}$ for $A \leftarrow A; P \cup \{Q\}$ is successful. By Proposition 5.4.8 and 5.4.17(2), A is never selected on $\mathcal{T}^s$. By Proposition 5.4.9(1), the P-tree $\mathcal{S}$ for $A \leftarrow Q; P \cup \{Q\}$ is successful, and A is never selected on $\mathcal{S}^s$. By Proposition 5.4.11, the P-tree for $A \leftarrow Q; P \cup \{A\}$ is successful.

- The last rule applied is MP# 1, so δ is of the following form:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ [P] \Rightarrow^* A & \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i\end{array}}{[A \leftarrow \mathbf{B}; P] \Rightarrow^* A}$$

Let, by the Inductive Hypothesis, for $i \in [1, n]$, $\mathcal{T}_i$ be successful and finite P-trees for $A \leftarrow A; P \cup \{B_i\}$. Let $\mathcal{T}$ be the P-tree for $A \leftarrow A; P \cup \{\mathbf{B}\}$. By Proposition 5.4.8 and 5.4.17, $\mathcal{T}$ is successful and finite and A is not selected on $\mathcal{T}$. By Proposition 5.4.9, $\mathcal{T}$ is also the P-tree for $A \leftarrow \mathbf{B}; P \cup \{\mathbf{B}\}$. Let $\mathcal{V}$ be the P-tree for $A \leftarrow \mathbf{B}; P \cup A$. Let, by the assumptions and the inductive hypothesis, $\mathcal{W}$ be the finite and successful P-tree for $P \cup \{A\}$. By Proposition 5.4.12, $\mathcal{V}$ is successful and finite.

- The last rule applied is MP# 2. This case is proven similar to the above case for MP# 1.
- The last rule applied is MP$^*_!$, so δ is of the following form:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow B_i & \sum_{i=1}^{m} [A \leftarrow A; P] \Rightarrow^* C_i\end{array}}{[A \leftarrow (\mathbf{B} \lhd !), !, (\mathbf{C} \lhd !); P] \Rightarrow^* A}$$

Let $Q_1 \in (\mathbf{B} \lhd !)$, and let $Q_2 \in (\mathbf{C} \lhd !)$. Let $R = A \leftarrow Q_1, !, Q_2; P$.

Let, by the Inductive Hypothesis, the P-trees for B_i w.r.t. $A \leftarrow A; P$ be successful. By Proposition 5.4.9 and 5.4.17, the P-tree $\mathcal{V}$ for $A \leftarrow A; P \cup \{Q_1\}$ is successful and A is never selected on $\mathcal{V}^s$.

By Proposition 5.4.6, the P-tree $\mathcal{T}$ for $R \cup \{Q_1\}$ is successful and A is never selected on $\mathcal{T}^s$.

Similarly, by the inductive hypothesis and the Propositions 5.4.8,5.4.17, and 5.4.6, the P-tree $R \cup \{Q_2\}$ is successful and finite and A is never selected on it. By the Propositions 5.4.16 and 5.4.15, the P-tree for $R \cup \{Q_1, !, Q_2\}$ is successful and finite, and A is never selected on it.

Therefore, by Proposition 5.4.19, the P-tree for $R \cup \{A\}$ is successful and finite.

- The last rule applied is $\mathrm{PFX}_{1!}$, so δ is of the following form:

$$\frac{\begin{array}{c}\vdots\\ [P] \Rightarrow A\end{array} \quad \begin{array}{c}\vdots\\ \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i\end{array} \quad \begin{array}{c}\vdots\\ [A \leftarrow A; P] \Rightarrow \neg D\end{array}}{[A \leftarrow \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow A}$$

  Let $R = A \leftarrow \mathbf{B}, D, Q; P$, where $Q \in (\mathbf{E} \lhd !)$.
  By derivability of the sequents $[A \leftarrow A; P] \Rightarrow^* B_i$, the inductive hypothesis and Proposition 5.4.17, the P-tree for $A \leftarrow A; P \cup \{\mathbf{B}\}$ is finite and successful.
  By derivability of the sequent $[A \leftarrow A; P] \Rightarrow \neg D$ and the Inductive Hypothesis, the P-tree for $A \leftarrow A; P \cup \{D\}$ is finitely failed.
  Thus, by Proposition 5.4.15, the P-tree for $A \leftarrow A; P \cup \{\mathbf{B}, D\}$ is finitely failed.
  Therefore, by Proposition 5.4.18, the P-tree for $A \leftarrow A; P \cup \{\mathbf{B}, D, Q\}$ is finitely failed.
  By derivability of $[P] \Rightarrow A$ and the Inductive Hypothesis, the P-tree for $P \cup \{A\}$ is successful.
  We can conclude, by Proposition 5.4.9 and 5.4.12(2), that the P-tree for $A \leftarrow \mathbf{B}, D, Q; P \cup \{A\}$ is successful.
- The last rule applied is $\mathrm{PFX}_{2!}$, so δ is of the following form:

$$\frac{\begin{array}{c}\vdots\\ [P] \Rightarrow \neg A\end{array} \quad \begin{array}{c}\vdots\\ \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow^* B_i\end{array} \quad \begin{array}{c}\vdots\\ [A \leftarrow A; P] \Rightarrow \neg D\end{array}}{[A \leftarrow \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

  Let $Q \in (\mathbf{E} \lhd !)$.
  This case is similar to the case $\mathrm{PFX}_{1!}$ above. The P-tree for $A \leftarrow A; P \cup \{A \leftarrow \mathbf{B}, D, Q\}$ is finitely failed.
  In the present case, by derivability of $[P] \Rightarrow \neg A$ and the Inductive Hypothesis, the P-tree for $P \cup \{A\}$ is finitely failed.
  By Proposition 5.4.12(4), the P-tree for $A \leftarrow \mathbf{B}, D, Q; P \cup \{A\}$ is finitely failed.
- The last rule applied is $\mathrm{PFX}_{\neg 2!}$, so δ is of the following form:

$$\frac{\begin{array}{c}\vdots\\ \sum_{i=1}^{n} [A \leftarrow A; P] \Rightarrow C_i\end{array} \quad \begin{array}{c}\vdots\\ \sum_{i=1}^{m} [A \leftarrow A; P] \Rightarrow^* B_i\end{array} \quad \begin{array}{c}\vdots\\ [A \leftarrow A; P] \Rightarrow \neg D\end{array}}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

  Let $Q_1 \in \mathbf{C} \lhd !$, and let $Q_2 \in \mathbf{E} \lhd !$.
  Like in the above case, we can infer that
  1. the P-tree for $A \leftarrow A; P \cup \{Q_1\}$ is successful;
  2. the P-tree for $A \leftarrow A; P \cup \{\mathbf{B}, D, Q_2\}$ is finitely failed.

  It follows from Proposition 5.4.20 that the P-tree for $A \leftarrow Q_1, \mathbf{B}, !, D, Q_2; P \cup \{A\}$ is finitely failed.
- The last rule applied is $\mathrm{PFX}_{3!}$. This case is similar to the above case of $\mathrm{PFX}_{2!}$. Proposition 5.4.12(2) and (3) are now used to infer that the P-tree for $A \leftarrow Q_1, \mathbf{B}, !, D, Q_2; P \cup \{A\}$ is finitely failed.

This concludes the proof of soundness of $\mathbf{S}_{Pr!}$. □

5.6 More useful properties of P-trees

We will prove completeness of $\mathbf{S}_{Pr!}$ by induction on a suitable measure on the class of successful and finite P-trees. We will define the *weight* of a P-tree in this class the tuple of its search weight and its construction weight, where the search weight is the number of pre-P-trees in the associated sequence of pre-P-trees necessary to constitute failure or success of the eventual limit-tree, and where the construction weight is the length of the associated sequence of pre-P-trees.

5.6.1. DEFINITION. Let $\mathcal{T}$ be a P-tree. Let $\mathcal{T}_1, \mathcal{T}_2, \ldots$ be the associated sequence of pre-P-trees. The *weight* of $\mathcal{T}$, $w(\mathcal{T})$, is the tuple

$$\langle sw(\mathcal{T}), \#(\mathcal{T})\rangle,$$

where

$$\begin{array}{rcll} \#(\mathcal{T}) & = & n & \text{if } n \text{ is minimal such that } \mathcal{T}_n = \mathcal{T} \\ & = & \omega & \text{otherwise} \\ sw(\mathcal{T}) & = & n \quad \text{if } \mathcal{T}_n = ext(\mathcal{T}^s) & \text{if } \mathcal{T} \text{ is successful} \\ & = & \#(\mathcal{T}) & \text{otherwise} \end{array} \qquad \square$$

5.6.2. PROPOSITION. *Let $\mathcal{T}$ be a* P*-tree. If $\mathcal{T}$ is successful, $sw(\mathcal{T})$ is finite, and $sw(\mathcal{T}) \leq \#(\mathcal{T})$. If $\mathcal{T}$ is finitely failed, $\#(\mathcal{T}) = sw(\mathcal{T}) < \omega$.*

Observe that both components of the weight can be infinite. However, in the completeness proof below, we will only be concerned with the weight of P-trees that are successful or finitely failed, that is, P-trees of which the search weight is finite. Thus the range of the weight function on the relevant domain is $\omega \times (\omega + 1)$. On the weight of P-trees we impose a partial order $\prec$, as follows:

5.6.3. DEFINITION. $\langle a, b\rangle \prec \langle c, d\rangle$ iff $a < c$, or $a = b < d < \omega$.

The order type of this order on the intended range of the weight function is $\omega \times (\omega + 1)$.

As a preparation to the completeness lemma, we now first state some useful propositions, which will be crucial in the proof of the completeness lemma. We omit most of the proofs.

5.6.4. PROPOSITION. *Let B be a pure atom, and Q a query. Let $\mathcal{V}$ be the successful* P*-tree for $P \cup \{Q, B\}$. Let $\mathcal{T}$ be the* P*-tree for $P \cup \{Q\}$. Let $\mathcal{S}$ be the* P*-tree for $P \cup \{B\}$. Then*

1. *$\mathcal{T}$ and $\mathcal{S}$ are successful*
2. *If A is never selected on $\mathcal{V}^s$, then A is never selected on $\mathcal{T}^s$ and $\mathcal{S}^s$.*
3. *$sw(\mathcal{T}) < sw(\mathcal{V})$, and $sw(\mathcal{S}) < sw(\mathcal{V})$.*
4. *If $\mathcal{V}$ is finite, then $\mathcal{T}$ and $\mathcal{S}$ are finite and $\#(\mathcal{T}) < \#(\mathcal{V})$, and $\#(\mathcal{S}) < \#(\mathcal{V})$.*
5. *If $\mathcal{V}$ is finite and A is never selected on $\mathcal{V}$, then A is never selected on $\mathcal{S}$ and $\mathcal{S}$.*

5.6.5. PROPOSITION. *Let $\mathcal{T}$ be the successful* P*-tree for $P \cup \{Q, !\}$. Let $\mathcal{S}$ be the* P*-tree for $P \cup \{Q\}$. Then*

1. *$\mathcal{T}$ is finite*
2. *$\mathcal{S}$ is successful*
3. *If A is never selected on $\mathcal{T}$, then A is never selected on $\mathcal{S}^s$.*
4. *$sw(\mathcal{S}) < sw(\mathcal{T})$.*

The above two propositions combine to the following more general case:

5.6.6. PROPOSITION. *Let $B_1, \ldots, B_n$ be pure atoms with $n > 1$, and let $Q \in (B_1, \ldots, B_n) \lhd\, !$. Let $\mathcal{T}$ be the successful* P*-tree for $P \cup \{Q\}$. Let, for $i \in [1, n]$, $\mathcal{T}_i$ be the* P*-tree for $P \cup \{B_i\}$. Then*

1. *The $\mathcal{T}_i$ are all successful,*
2. *$sw(\mathcal{T}_i) < sw(\mathcal{T})$,*
3. *If A is not selected on $\mathcal{T}^s$, then A is not selected on any of the $\mathcal{T}_i^s$.*

5.6.7. PROPOSITION. *Let Q be a query and let $\mathcal{T}$ be the successful and finite* P*-tree for $P \cup \{Q\}$. Then one of the following cases applies.*

1. *There are atomic $B_1, \ldots, B_n$ such that $Q = B_1, \ldots, B_n$. The* P*-trees $\mathcal{T}_i$ for $P \cup \{B_i\}$ are successful and finite. Also, $sw(\mathcal{T}_i) \leq sw(\mathcal{T})$. If A is never selected on $\mathcal{T}$ then A is never selected on any of the $\mathcal{T}_i$.*
2. *There are atomic $C_1, \ldots, C_n$ and $B_1, \ldots, B_m$ and a query Q', such that $Q_1 \in ((C_1, \ldots, C_n) \lhd\, !)$, and $Q = Q_1, !, B_1, \ldots, B_m$. The* P*-trees $\mathcal{T}_i$ for $P \cup \{C_i\}$ are successful and the* P*-trees $\mathcal{S}_i$ for $P \cup \{B_i\}$ are successful and finite. Also, $sw(\mathcal{T}_i) < sw(\mathcal{T})$, and $sw(\mathcal{S}_i) \leq \#(\mathcal{S}_i) < sw(\mathcal{T})$. If A is never selected on $\mathcal{T}$, then A is never selected on any of the $\mathcal{T}_i^s$ and $\mathcal{S}_i$.*

PROOF. By the above Propositions 5.6.4, 5.6.5, and 5.6.6. □

5.6.8. PROPOSITION. *Let Q be a query and let $\mathcal{T}$ be the finitely failed* P*-tree for $P \cup \{Q\}$. Then there are Q_1, D, Q_2 (with Q_1 and Q_2 possibly empty), such that*

1. *$Q = Q_1, D, Q_2$.*
2. *The search tree $\mathcal{S}$ for $P \cup \{D\}$ is finitely failed and $sw(\mathcal{S}) \leq sw(\mathcal{T})$.*
3. *The search tree $\mathcal{V}$ for $P \cup \{Q_1\}$ is successful and finite and $sw(\mathcal{V}) < sw(\mathcal{T})$.*
4. *If A is not selected on $\mathcal{T}$ then A is not selected on $\mathcal{S}$ and $\mathcal{V}$.*

The following construction will be used in the proof of the completeness lemma.

Let $\mathcal{T}$ be the successful or finitely failed P-tree for $A \leftarrow Q; R \cup \{A\}$.
We split $\mathcal{T}$ into two relevant semi-P-trees $\mathcal{T}_L$ and $\mathcal{T}_R$, as follows:

$\mathcal{T}_L$ is the subtree of $\mathcal{T}$ generated by the leftmost immediate descendant of the root A. Thus, $\mathcal{T}_L$ is the P-tree for $P \cup \{Q\}$.
$\mathcal{T}_R$ is the result of deleting $\mathcal{T}_L$ from $\mathcal{T}$.

Observe that $\mathcal{T}_R$ is not necessarily a P-tree.

We now prove some propositions which will be useful in the proof of the completeness lemma. Again, we omit the proofs.

5.6.9. PROPOSITION.

1. *$\mathcal{T}_L$ is either successful or finitely failed.*

2. *If $\mathcal{T}$ is finitely failed, then $\mathcal{T}_L$ is finitely failed.*
3. *If $\mathcal{T}$ is finite, then $\mathcal{T}_L$ and $\mathcal{T}_R$ are finite.*
4. $sw(\mathcal{T}_L) < sw(\mathcal{T})$.

5.6.10. PROPOSITION. *If $\mathcal{T}$ is successful but not finite, then one of the following cases holds:*

1. *$\mathcal{T}_L$ is successful, $sw(\mathcal{T}_L) < sw(\mathcal{T})$, and A is not selected on $\mathcal{T}_L^s$.*
2. *$\mathcal{T}_L$ is finitely failed, $sw(\mathcal{T}_L) = \#(\mathcal{T}_L) < sw(\mathcal{T})$, and A is not selected on $\mathcal{T}_L$.*

In addition, we need the following two P-trees $\mathcal{L}$ and $\mathcal{R}$, which are closely related to $\mathcal{T}_L$ and $\mathcal{T}_R$.

$\mathcal{L}$ is the P-tree for $A \leftarrow A; R \cup \{Q\}$,
$\mathcal{R}$ is the P-tree for $R \cup \{A\}$.

5.6.11. PROPOSITION.

1. *$\mathcal{L}$ is successful iff $\mathcal{T}_L$ is successful.*
2. *$\mathcal{L}$ is finitely failed iff $\mathcal{T}_L$ is finitely failed.*
3. *If $\mathcal{L}$ is successful, then $\mathcal{L}^s = \mathcal{T}_L^s$.*
4. *If $\mathcal{T}_L$ is finite, then $\mathcal{L} = \mathcal{T}_L$.*
5. $sw(\mathcal{L}) < sw(\mathcal{T})$.

PROOF. The left-to-right direction of both 1) and 2) follows from Proposition 5.4.9. By Proposition 5.4.5, A is not selected on $\mathcal{T}_L^s$ if $\mathcal{T}_L$ is successful, and A is not selected on $\mathcal{T}_L$ if $\mathcal{T}_L$ is finitely failed. The right-to-left direction of both 1) and 2) now follows from Proposition 5.4.6 and 5.4.7.
3) and 4) follow from Proposition 5.4.6 and 5.4.9. For 5), observe the following. By Proposition 5.6.9(1), $\mathcal{T}_L$ is either successful or finitely failed. Therefore, by Proposition 5.6.10, $sw(\mathcal{L}) \leq sw(\mathcal{T}_L)$. Thus, by 5.6.9(4), $sw(\mathcal{L}) < sw(\mathcal{T})$. □

5.7 Completeness

We are now in position to state and prove the completeness lemma for $\mathbf{S}_{Pr!}$.

5.7.1. LEMMA.

1. *If the* P*-tree for $P \cup \{A\}$ is successful then $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow A$.*
2. *If the* P*-tree for $P \cup \{A\}$ successful and finite then $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow^* A$.*
3. *If the* P*-tree for $P \cup \{A\}$ finitely failed then $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow \neg A$.*

PROOF. By induction on the weight of the relevant P-trees.

The base case, $w(\mathcal{T}) = \langle 2, 2\rangle$, corresponds to the axioms for immediate success of the empty goal and the axiom for immediate failure of undefined atoms.

(IH) Suppose the lemma holds for P-trees $\mathcal{T}$ with $w(\mathcal{T}) <_o \langle k, m\rangle$.

Let $\mathcal{T}$ be a successful or finitely failed P-tree for $P \cup \{A\}$, with $w(\mathcal{T}) = \langle k, m \rangle$. If $P_A = \emptyset$ or A is the empty goal, we are in the base case. Thus, without loss of generality we can assume, by the rules HP and HP*, that $P = A \leftarrow Q; R$ for some R.

Let $\mathcal{T}$ be a successful or finitely failed P-tree for $A \leftarrow Q; R \cup \{A\}$. We split $\mathcal{T}$ into two relevant semi-P-trees $\mathcal{T}_L$ and $\mathcal{T}_R$, as above in Section 5.6. Also, let $\mathcal{L}$ and $\mathcal{R}$ be as defined in Section 5.6.

We distinguish the following cases.

- $\mathcal{T}$ is not finite.
  Observe that $\mathcal{T}$ must be successful in this case.
  By Proposition 5.4.4, A is not selected in $\mathcal{T}^s$, except in the root.
  Using Proposition 5.6.10, we distinguish the following cases.
  - $\mathcal{T}_L$ is successful. By Proposition 5.6.10 and 5.6.11(1) and (3), $\mathcal{L}$ is successful, $sw(\mathcal{L}) = sw(\mathcal{T}_L) < sw(\mathcal{T})$, and A is never selected on $\mathcal{L}^s$.
    Let $Q \in (B_1, \ldots, B_n) \lhd\, !$. Let, for $i \in [1, n]$, $\mathcal{L}_i$ be the P-trees for $A \leftarrow A; R \cup \{B_i\}$.
    By Proposition 5.6.6 and 5.4.6, the $\mathcal{L}_i$ are successful and $sw(\mathcal{L}_i) \leq sw(\mathcal{L})$. Thus, $sw(\mathcal{L}_i) < sw(\mathcal{T})$, and the Inductive Hypothesis applies to the $\mathcal{L}_i$. Therefore, the sequents $[A \leftarrow A; R] \Rightarrow B_i$ are derivable. Thus, by an application of MP$_!$, the sequent $[P] \Rightarrow A$ is derivable.
  - $\mathcal{T}_L$ is finitely failed. Because $\mathcal{T}$ is successful, $\mathcal{T}^s$ properly extends $A \uplus \mathcal{T}_L$. By Proposition 5.6.11(2),(4),and (5), $\mathcal{T}_L = \mathcal{L}$, $\mathcal{L}$ is finitely failed, and $sw(\mathcal{L}) < sw(\mathcal{T})$.
    By Proposition 5.6.8, there are queries Q_1, Q_2, and a pure atom D such that $Q = Q_1, D, Q_2$, such that
    a. the P-tree $\mathcal{S}$ for $A \leftarrow A; R \cup \{D\}$ is finitely failed and $sw(\mathcal{S}) \leq sw(\mathcal{L}) < sw(\mathcal{T})$, and
    b. the P-tree $\mathcal{V}$ for $A \leftarrow A; R \cup \{Q_1\}$ is successful and finite, and $sw(\mathcal{V}) < sw(\mathcal{L}) < sw(\mathcal{T})$.
    The Inductive Hypothesis applies to $\mathcal{S}$, so the sequent $[A \leftarrow A; R] \Rightarrow \neg D$ is derivable in $\mathbf{S}_{P_r!}$.
    By Proposition 5.4.20, ! cannot occur in Q_1, as $\mathcal{T}$ is successful. Thus, $Q_1 = B_1, \ldots, B_n$, where the B_i are pure atoms. By Proposition 5.6.7(1), the P-trees $\mathcal{V}_i$ for $A \leftarrow A; R \cup B_i$ are successful and finite, and $sw(\mathcal{V}_i) \leq sw(\mathcal{V})$. Therefore, $sw(\mathcal{V}_i) < sw(\mathcal{T})$. The Inductive Hypothesis applies to the $\mathcal{V}_i$, so the sequents $[A \leftarrow A; R] \Rightarrow^* B_i$ are derivable in $\mathbf{S}_{P_r!}$.
    Furthermore, consider the semi-P-tree $\mathcal{R}'$, obtained by restricting $\mathcal{T}^s$ to $\mathcal{T}_R$. A is never selected on $\mathcal{R}'$, except in the root, so by Proposition 5.4.6, $\mathcal{R}^s = \mathcal{R}'$. In particular, $\mathcal{R}$ is successful and $sw(\mathcal{R}) < sw(\mathcal{T})$. Thus the Inductive Hypothesis applies to $\mathcal{R}$, and we infer that the sequent $[R] \Rightarrow A$ is derivable in $\mathbf{S}_{P_r!}$.
    By an application of PFX$_{1!}$, $[P] \Rightarrow A$ is derivable.

- $\mathcal{T}$ is finite. By Proposition 5.4.4, A is not selected in $\mathcal{T}$, except in the root. By Proposition 5.6.9(3) $\mathcal{T}_R$ and $\mathcal{T}_L$ are finite. Also, by Proposition 5.6.11, $\mathcal{L} = \mathcal{T}_L$. We distinguish the following cases:
  - $\mathcal{T}_L$ is successful.
    By Proposition 5.6.7, we can distinguish the following two cases:
    * [C1] $Q = B_1, \ldots, B_n$, where the B_i are pure atoms. By Proposition 5.6.7(1), 5.6.11(5), the starsequents $[A \leftarrow A; R] \Rightarrow^* B_i$ are derivable for $i \in [1, n]$.
      Also, because the Prolog cut ! does not occur in Q, and because A is never selected on $\mathcal{T}$, $\mathcal{T}_R = \mathcal{R}$. Thus $\mathcal{R}$ is finite. We distinguish two cases:
      · [C1a] $\mathcal{R}$ is successful. Observe that $sw(\mathcal{R}) \leq \#(\mathcal{R}) < \#(\mathcal{T})$. Thus the Inductive Hypothesis applies to $\mathcal{R}$, so $[R] \Rightarrow^* A$ is derivable.
      By an application of MP# 1 the sequent $[P] \Rightarrow^* A$ is derivable.
      · [C1b] $\mathcal{R}$ is finitely failed. Observe that $sw(\mathcal{R}) = \#(\mathcal{R}) < \#(\mathcal{T})$. Thus the Inductive Hypothesis applies to $\mathcal{R}$, and $[R] \Rightarrow \neg A$ is derivable.
      By an application of MP# 2 the sequent $[P] \Rightarrow^* A$ is derivable.
    * [C2] $Q = Q_1, !, C_1, \ldots, C_m$, where $Q_1 \in (B_1, \ldots, B_n \lhd !)$, for pure atoms C_j and B_i.
      By Proposition 5.6.11, 5.6.7, and the Inductive Hypothesis, it follows that the sequents $[A \leftarrow A; R] \Rightarrow B_i$ are derivable for $i \in [1, n]$, and the starsequents $[A \leftarrow A; R] \Rightarrow^* C_i$ are derivable for $i \in [1, m]$.
      By an application of $\mathrm{MP}^*_!$ the sequent $[P] \Rightarrow^* A$ is derivable.
  - $\mathcal{T}_L$ is finitely failed.
    By Proposition 5.6.8, there are Prolog queries Q'_1 and Q'_2, and a pure atom D, such that $Q = Q'_1, D, Q'_2$. Also, let $\mathcal{S}$ be the P-tree for $A \leftarrow A; R \cup \{D\}$ and let $\mathcal{V}$ be the P-tree for $A \leftarrow A; R \cup \{Q'_1\}$. $\mathcal{S}$ is finitely failed and $sw(\mathcal{S}) < sw(\mathcal{T})$, while $\mathcal{V}$ is successful and finite and $sw(\mathcal{V}) < sw(\mathcal{T})$. Thus, the Inductive Hypothesis applies to $\mathcal{S}$, and we infer that
    $[A \leftarrow A; R] \Rightarrow \neg D$ is derivable.
    According to Proposition 5.6.7, we can distinguish the following two cases for Q'_1:
    * $Q'_1 = B_1, \ldots, B_n$, where the B_i are all atomic. As in this case [C1] above, it follows that the starsequents $[A \leftarrow A; R] \Rightarrow^* B_i$ are derivable for $i \in [1, n]$.
      Also, because the Prolog cut ! does not occur in Q'_1, and because A is never selected on $\mathcal{T}$, $\mathcal{T}_R = \mathcal{R}$. Again, we distinguish the following two cases:
      · $\mathcal{R}$ is successful. As in the case [C1a], $[R] \Rightarrow^* A$ is derivable. By an application of $\mathrm{PFX}_{3!}$, the starsequent $[P] \Rightarrow^* A$ is derivable.
      · $\mathcal{R}$ is finitely failed. As in the case [C1b], $[R] \Rightarrow \neg A$ is derivable. By an application of $\mathrm{PFX}_{2!}$ the starsequent $[P] \Rightarrow^* A$ is derivable.

* $Q' = Q_1, !, B_1, \ldots, B_m$, where $Q_1 \in (C_1, \ldots, C_n \lhd !)$, for pure atoms C_j and B_i.
  Again, as in the case [C2], the sequents $[A \leftarrow A; R] \Rightarrow B_i$ are derivable for $i \in [1, n]$, and the starsequents $[A \leftarrow A; R] \Rightarrow^* C_i$ are derivable for $i \in [1, m]$. By an application of $\mathrm{PFX}_{\neg 2!}$, the sequent $[P] \Rightarrow \neg A$ is derivable.

This completes the proof of the completeness lemma. □

The correctness theorem for $\mathbf{S}_{Pr!}$ (Theorem 5.2.1) now immediately follows from Lemmas 5.3.12, 5.5.1, and 5.7.1.

It is not difficult to see that $\mathbf{S}_{Pr!}$-derivations of sequents $[P] \Rightarrow L$, for pure Prolog programs P, are in fact $\mathbf{S}^-_{Pr}$-derivations. Conversely, every $\mathbf{S}^-_{Pr}$-derivation is also an $\mathbf{S}_{Pr!}$-derivation. By the equivalence of $\mathbf{S}_{Pr}$ and $\mathbf{S}^-_{Pr}$ we have the following, not very surprising, result:

5.7.2. PROPOSITION. *Let P be a pure Prolog program. Then*

1. $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow A$ *iff* $\mathbf{S}_{Pr} \vdash [P] \Rightarrow A$.
2. $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow \neg A$ *iff* $\mathbf{S}_{Pr} \vdash [P] \Rightarrow \neg A$.

5.8 Alternative versions of the Prolog cut

As observed in Section 5.1, the standard Prolog cut can be thought of as the composition of two simpler pruning operators: the soft cut and the one solution operator.

The soft cut is a restricted version of the standard cut, which only prevents backtracking on its origin. That is, a soft cut commits to the choices made for the parent goal (the origin), but it does not commit to the choices made for body atoms to its left.

The soft cut can be axiomatised in a variant of the calculus $\mathbf{S}_{Pr!}$, obtained by replacing the cut-specific rules by appropriate variants.

To shorten notation, we write, in the rules below, $[R] \Rightarrow \mathbf{B}$ and $[R] \Rightarrow^* \mathbf{B}$ for, respectively, $\sum_{i=1}^n [R] \Rightarrow B_i$ and $\sum_{i=1}^n [R] \Rightarrow^* B_i$.

* the rule $\mathrm{MP}^*_!$ is replaced by the following rule $\mathrm{MP}^{*h}_!$:

$$\frac{[A \leftarrow A; P] \Rightarrow^* \mathbf{B} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{C}}{[A \leftarrow (\mathbf{B} \lhd !), !, \mathbf{C}; P] \Rightarrow^* A}$$

* In addition, the rule $\mathrm{PFX}_{\neg 2!}$ is replaced by the following variant $\mathrm{PFX}_{\neg 2!}{}^h$:

$$\frac{[A \leftarrow A; P] \Rightarrow^* \mathbf{C} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{B} \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

The rationale for these two variant rules is the following: As the soft cut does not prevent backtracking on body atoms to its left, it is not sufficient that the respective searches for these atoms succeed – during backtracking via these atoms, the search might still diverge. Thus, both in the modus ponens and in the prefixing rule specific for the soft cut, the leftmost assumptions, dealing with the body atoms that lie to the left of the cut, need to be starsequents instead of non-star sequents.

The one solution cut does not prevent backtracking on its origin, but instead, it prevents backtracking via the earlier body atoms. That is, it does not prune on the origin of the cut, but only below, between the origin and the activated cut.

This "body-cut" can also be axiomatised by a variant on $\mathbf{S}_{P_r!}$, again obtained by modifying the cut-specific rules $\mathrm{MP}^*_!$ and $\mathrm{PFX}_{\neg 2!}$:

- The rule $\mathrm{MP}^*_!$ is replaced by *two* variants:

$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow \mathbf{B} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{C}}{[A \leftarrow (\mathbf{B} \lhd !), !, \mathbf{C}; P] \Rightarrow^* A}$$

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow \mathbf{B} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{C}}{[A \leftarrow (\mathbf{B} \lhd !), !, \mathbf{C}; P] \Rightarrow^* A}$$

- The rule $\mathrm{PFX}_{\neg 2!}$ is repaced by the following three variants:

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow \mathbf{C} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{B} \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow \neg A}$$

$$\frac{[P] \Rightarrow A \quad [A \leftarrow A; P] \Rightarrow \mathbf{C} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{B} \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow A}$$

$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow \mathbf{C} \quad [A \leftarrow A; P] \Rightarrow^* \mathbf{B} \qquad [A \leftarrow A; P] \Rightarrow \neg D}{[A \leftarrow (\mathbf{C} \lhd !), !, \mathbf{B}, D, (\mathbf{E} \lhd !); P] \Rightarrow^* A}$$

The rationale for the three prefixing variants is the following: When the search for A via the first A-clause fails, the search is continued via further A-clauses. A processed one solution operator in the body of the first A-clause does not (unlike the standard cut and the soft cut) prevent this backtracking. Then there are three possibilities: the search for A eventually succeeds (finitely or not), or it fails finitely. Each of these possibilities is reflected in a prefixing rule. The rationale for the two variants of the modus ponens rule is similar.

While the standard Prolog cut is in fact the composition of the above two simpler pruning operations, the calculus $\mathbf{S}_{P_r!}$ is clearly *not* the simple union of the above two calculi corresponding to its two components. The reason is that (unlike the connectives we discussed in Section 3.8) the pruning operators have a global, rather than a local effect.

Several alternatives for the standard Prolog cut have been suggested and studied, often motivated by the fact that the use of the Prolog cut may lead to a loss of soundness and completeness with respect to the completion semantics. In particular, a generalisation of the commit operators of the concurrent logic programming languages has been studied extensively in [HLS90], which does not have this disadvantage and which in addition, unlike the Prolog cut, behaves well under partial evaluation. It is unclear whether any of these alternative pruning operators can, to the full extent, be incorporated in the Gentzen style format. Certainly simple commit-operators on atoms (like commit$_1$, pruning alternative solutions on just one body atom in a clause) are amenable to our approach; however, we conjecture that more involved versions, like those suggested in [HLS90], would lead to an explosion of derivation rules.

5.9 Right monotonicity of cut

By inspection of the rules of $\mathbf{S}_{Pr!}$, it is clear that in some cases cuts can be added to the clauses of a program without changing its procedural behaviour. In particular, if the body of a clause contains a cut, any cuts can be added to its right side. That is, the following rule (and also the star version) is admissible for $\mathbf{S}_{Pr!}$:

$$\frac{[P; A \leftarrow Q_1, !, Q_2, Q_3; R] \Rightarrow L}{[P; A \leftarrow Q_1, !, Q_2, !, Q_3; R] \Rightarrow L}$$

A proof of the soundness of the above rule, expressing the right monotonicity of the Prolog cut, proceeds by induction on depth of derivations, and uses the following fact:

5.9.1. LEMMA. *If* $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow^* A$ *then* $\mathbf{S}_{Pr!} \vdash [P] \Rightarrow A$.

In contrast, the reverse direction of this rule is not sound. That is, if the body of a clause contains two cuts, the rightmost occurrence of the cut cannot be deleted without altering the procedural behaviour of the program. A counterexample is the following program P:

$$\begin{array}{l} A \leftarrow B, !, C, !, D \\ B \leftarrow \\ C \leftarrow \\ C \leftarrow C \end{array}$$

A finitely fails from P. However, if the first clause in P is replaced by the clause $A \leftarrow B, !, C, D$, then a Prolog search for A diverges via the reflexive clause $C \leftarrow C$.

Also, the Prolog cut is not 'left-monotonous'. That is, while cuts can be added to the right side of an already present cut, they cannot safely be added to its left side. A counterexample is the following program:

$$\begin{array}{l} A \leftarrow B, D, !, C \\ A \leftarrow \\ B \leftarrow \end{array}$$

A succeeds on this program. In contrast, if the first clause is replaced by the clause $A \leftarrow B, !D, !, C$, then A finitely fails from the resulting program.

A further analysis of the calculus $\mathbf{S}_{Pr!}$ might reveal other useful admissible rules involving cut.

5.10 Notes

- Our approach and results should be compared with the Mints deductive calculus introduced in [Min90] and extended in [She92]. A difference between our approach and the Mints calculus is that the Mints calculi directly axiomatise the way in which an LD(NF) tree is built up for a single program. The procedural Gentzen calculi have a slightly dynamic flavour; the program is built up in the course of a derivation. Another difference is that the Gentzen calculus $\mathbf{S}_{Pr!}$, other than the Mints calculus, is restricted to the propositional case. This restriction is not necessary in principle: the calculi $\mathbf{S}_{Pr}$ and $\mathbf{S}_{Pr!}$ can be extended to deal with predicate programs, incorporating unification and computed answer substitution. However, the predicate versions of these calculi seem to require too many derivation rules (each a version however of the original $\mathbf{S}_{Pr}$ and $\mathbf{S}_{Pr!}$ rules) to be interesting. Nevertheless we want to stress that the restriction to propositional programs is by no means essential in the Gentzen format.
- The standard interpretation of negation in Logic Programming, negation as finite failure, is implemented in Prolog by the cut-fail definition. Apt and Teusink [AT95] have shown, using P-trees to model the Prolog left to right and top down, computation procedure, that this implementation is indeed correct for left-terminating programs (programs for which every LDNF-derivation is finite), in the sense that the cut-fail interpretation and the negation as failure interpretation yield the same set of computed answer substitutions. This result allows for transfer of various results on general logic programs to their Prolog counterparts. In particular, it allows for the construction of the declarative interpretation of pure Prolog programs involving negated body atoms and the investigation of their correctness. Further investigations by Teusink have shown that the results also hold for programs with possibly infinite LDNF-derivations. An obvious question, which might be interesting for further research, is whether these results (or rather, the propositional versions of these results) can also be obtained by relating the Gentzen calculus $\mathbf{S}_{Pr!}$ to the version of $\mathbf{S}_{Pr}$ appropriate for normal programs.

Discussion

It is Logic Programming folklore that, although in theory the declarative and procedural interpretation of logic programs coincide, this correspondence only holds for abstract interpreters, while under execution of programs in practice, for instance in Prolog, these two interpretations diverge. In the previous two chapters we have studied the effects of computation styles, in particular of top-down clause processing. Using a Gentzen style sequent calculus format, we have studied several computation styles using top-down clause processing as substructural logics.

The analysis of search procedures as substructural logics reveals the following general picture:

- top-down clause processing is reflected in basic modus ponens and prefixing rules, dealing with success via the top-most relevant clause (modus ponens rules) and failure via the top-most relevant clause (prefixing rules);
- selection rules, specifying the order of processing of composite goals, are reflected in introduction rules for (procedural) connectives on goals — alternatively, introduction rules specifying a particular selection rule, can be incorporated in the relevant modus ponens and prefixing rules;
- extensive backtracking, such as in Prolog style computation, is reflected in refined versions of the modus ponens and prefixing rules.

In addition, extra connectives on goals can be incorporated in any of the calculi, effecting a local change in the overall selection rule. The axiomatisation of Prolog with the Prolog cut shows that the format is also amenable to incorporation of operators with a more global effect.

As logics, the above procedural calculi seem to form a deviant class of non-monotonic substructural logics: most non-monotonic logics are supra-normal, deriving more than classical logic, while the above calculi are infra-normal, deriving less than classical logic. In this sense, the above procedural calculi are new and deviant examples of substructural logics, with unusual substructural rules and illustrating new substructural principles.

Unfortunately, our approach is rather isolated. There seem to be no obvious links between these procedural calculi and other classes of non-standard logics, and the semantics used is not very abstract. Even a direct correspondence to the Mints calculi, which axiomatise Prolog computation, has not yet been established. However, the robustness of several core-elements of the procedural calculi (modus ponens and prefixing rules corresponding to top-down clause processing; connectives corresponding to selection rules) suggests that, eventually, a more general picture might arise. Development of a more abstract semantics could be a useful part of future resaerch.

The issue of rule completeness was not addressed in the above two chapters. That is: which class of rules are admissible or derivable for any of the procedural calculi. It may very well be that any method appropriate to answer this question, also gives some useful insight in the general character of these calculi.

A drawback of the above approach is the relative technicality. The various soundness and completeness proofs do not seem to give any additional insights over informal correctness arguments. Still, the formal correctness proofs are not superfluous: experience has shown that it is only too easy, in a procedural context, to overlook cases.

An important issue in the evaluation of the above approach is its usefulness. As derivations in the various calculi studied tend to be relatively long, they do not seem to be very useful in establishing the outcome of computations on programs. In addition, derivations in the respective calculi establish success of single atoms from single programs, while in general the interest is in the correctness of a program with respect to a set of atomic queries. Also, in many cases, in practice it seems to be more easy to reason on search trees than on derivations in the relevant calculus. However, there are some exceptions. The right-monotonicity of the Prolog cut, for example, was established by inspection of the rules of the relevant calculus. The interest of the procedural calculi developed is may well lay in their usefulness as tools for analysis (witnessed for example by the alternatives for the Prolog cut, which naturally emerge in the procedural format, and by the various results on calculus for frugal Prolog), rather than in their use as alternative tools for computation.

Bibliography

[And93] J.H. Andrews. A logical semantics for depth-first prolog with ground negation. In D. S. Warren, editor, *Proceedings ICLP '93*. MIT Press, 1993.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[AT95] K.R. Apt and F. Teusink. Comparing negation in logic programming and in Prolog. In K.R.Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 111 - 133. The MIT Press, 1995.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[BT95] A. Brogi and F. Turini. Meta-logic for program composition: semantics issues. In K.R.Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 83 - 110. The MIT Press, 1995.

[Cer92] S. Cerrito. A linear axiomatization of negation as failure. *Journal of Logic Programming*, 12(1 & 2):1–24, 1992.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[Doe94] H. C. Doets. *From Logic to Logic Programming*. MIT Press, 1994.

[Dos92] K. Dosen. Modal logic as metalogic. *Journal of Logic, Language, and Information*, 1(3):173 - 202, 1992.

[FLMP93] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 102(1):86–113, 1993.

[Gab92] D. Gabbay. Metalevel features in the object level: modal and temporal logic programming III. In L.Farinas del Cerro and M. Penttonen, editors, *Intensional logics for programming*, pages 85–124. Clarendon Press, 1992.

[GLT90] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1990.

[HL88] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H.D. Abramson and M.H. Rogers, editors, *Proceedings of the Meta88 Workshop*, pages 23–52. MIT Press, 1988.

[HLS90] P.M. Hill, J.W. Lloyd, and J.C. Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(3):99 - 143, 1990.

[Isr92] D. Israel. The role(s) of logic in artificial intelligence. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (Vol I)*, pages 1 - 27. Clarendon Press, 1992.

[Jia94a] Y. Jiang. Ambivalent logic as the semantic basis for metalogic programming: I. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, June 1994.

[Jia94b] Y. Jiang. Ambivalent logic as the semantic basis of metalogic programming: Theory and practice. Technical report, Imperial College, Dept. of Computing, 1994.

[Kal] M. Kalsbeek. Computations and operators. manuscript.

[Kal93] M. Kalsbeek. The Vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.

[Kal94] M. Kalsbeek. Gentzen systems for logic programming styles. Technical Report CT-94-12, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1994.

[Kal95a] M. Kalsbeek. Correctness of the Vanilla meta-interpreter and ambivalent syntax. In K.R.Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 1 - 26. The MIT Press, 1995.

[Kal95b] M. Kalsbeek. Gentzen systems for logic programming styles i: Substructural aspects. *Bulletin of the IGPL*, 1995.

[Kal95c] M. Kalsbeek. Gentzen systems for logic programming styles ii: Logics for prolog. *Bulletin of the IGPL*, 1995.

[KJ95] M. Kalsbeek and Y. Jiang. A vademecum of ambivalent logic. In K.R.Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 27 - 56. The MIT Press, 1995.

[KK91] R.A. Kowalski and J. Kim. A metalogic programming approach to multi-agent knowledge and belief. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 231–246. Academic Press, 1991.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[LR93] G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Proceedings ICLP '93*, pages 354–373. MIT

Press, 1993.

[Mak93] D. Makinson. General Patterns in Nonmonotonic Reasoning. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 2, Nonmonotonic and Uncertain Reasoning*, chapter 2.2. Oxford University Press, 1993.

[Mil89] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 - 108, 1989.

[Min90] G.E. Mints. Several formal systems of the logic programming. *Computers and Artificial Intelligence*, 9:19 - 41, 1990.

[MS95a] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K.R.Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57 - 82. The MIT Press, 1995.

[MS95b] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

[Ric74] B. Richards. A point of reference. *Synthese*, 28:431–445, 1974.

[Sch89] U. Schöning. *Logic for Computer Scientists*, volume 8 of *Progress in Computer Science and Applied Logic.* Birkhauser, 1989.

[She92] J.C. Shepherdson. Mints type deductive calculi for logic programming. *Annals of Pure and Applied Logic*, 56:7 - 17, 1992.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Stä] R. Stärk. A transformation of propositional prolog programs into classical logic.

[Stä94] R. Stärk. The declarative semantics of the Prolog selection rule. In *9th Annual Symposium on Logic in Computer Science*, pages 252 - 261, Paris, 1994.

[Urq86] A. Urquhart. Many-valued logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic (Vol III)*, pages 71 - 116. Reidel, 1986.

[vB92] van Benthem. Logic as programming. *Fundamenta Informaticae*, 17(4):285 - 318, 1992.

Samenvatting

In deze dissertatie worden enkele logische aspecten van het logisch programmeren onderzocht. De drie delen waaruit dit proefschrift bestaat kunnen onafhankelijk van elkaar gelezen worden.

Ofschoon het logisch programmeren gebaseerd is op (een fragment van) de eerste orde predicaatlogica (FOL) wordt in de praktijk veelal gewerkt in extensies van FOL waarin het gebruikelijke syntactische onderscheid tussen predicaten, functies, formules en termen vervaagt of verdwijnt. Voorbeelden zijn Prolog, waar het voorkomen van variabelen op atoom-posities is toegestaan, en meta-logisch programmeren met de Vanilla meta-interpreter, waar atomen op term-posities worden gebruikt. In het eerste hoofdstuk rechtvaardigen we deze praktijk.

We definieren Ambivalente Logica AL, die verkregen wordt uit FOL door restricties op de standaard syntax te laten vervallen. AL kan beschouwd worden als een conservatieve extensie van FOL. Met een serie formele resultaten laten we zien dat AL een flexibele omgeving vormt voor de verschillende syntactisch ambivalente talen die gebruikt worden in Prolog, meta-logisch programmeren en formalisaties van kennis en geloof. We voorzien AL van een gesloten term semantiek, met betrekking tot welke een standaard afleidings-calculus gezond en volledig is. Verder laten we zien dat het gebruikelijke Martelli-Montanari algorithme aangepast kan worden voor AL, met behoud van de gewenste gebruikelijke eigenschappen. In combinatie met enkele andere bewijstheoretische basisresultaten laat dit zien dat resolutie een gezonde en volledige afleidingsstrategie is voor AL.

Het tweede hoofdstuk behandelt een klassiek probleem uit de grondslagen van het meta-logisch programmeren: de rechtvaardiging van de formeel incorrecte (ongetypeerde) Vanilla meta-interpreter. Verschillende benaderingen van het probleem worden beschouwd en vergeleken: procedurele en declaratieve correctheid, het gebruik van S-semantiek, het gebruik van de correcte (getypeerde) versie, de restrictie tot language independent object programma's. We argumenteren dat Hill en Lloyd's klassieke bewijs van de procedurele correctheid van de getypeerde interpreter grote algemeenheid

heeft. We geven een gedetailleerd bewijs voor de declaratieve correctheid van Vanilla met ambivalente syntax als onderliggende taal — ambivalente syntax is de geeigende syntax voor Vanilla, in het bijzonder voor de in de praktijk gangbare geamalgameerde extensies. Dit bewijs generaliseert vrijwel onmiddelijk voor amalgamaties van object programma met de gerelateerde Vanilla meta-interpreter.

In het derde deel worden procedurele calculi ontwikkeld en bestudeerd, die de effecten van diverse zoekmechanismen in implementaties van logisch programmeren, zoals Prolog, karakteriseren. Ofschoon volgens de theorie van het logisch programmeren de geintendeerde (declaratieve) betekenis van een programma overeenkomt met de procedurele, hoeft dit niet het geval te zijn voor geimplementeerde programma's. Zowel de zoekregel van een implementatie (de volgorde waarin de regels van een programma worden gebruikt) als de backtracking en de selectieregel (de volgorde waarin de onderdelen van een samengesteld doel worden behandeld) kunnen invloed hebben op de computationele uitkomst. De effecten van zoekmechanismen blijken gekarakteriseerd te kunnen worden met behulp van substructurele calculi in Gentzen formaat. Deze procedurele calculi hebben als karakteristieke expressies sequenten van de vorm $[P] \Rightarrow A$, die uitdrukken dat A slaagt uit het programma P onder het bedoelde zoekmechanisme. Voor de onderhavige zoekmechanismen relevante delen van de zoekbomen functioneren als semantiek voor de calculi en worden gebruikt voor het bewijzen van correctheidsresultaten.

In het derde hoofdstuk wordt allereerst een calculus besproken die overeenkomt met de standaard 'top-down' zoekregel en een ongespecificeerde selectieregel (depth first search). Negation as failure kan in dit formaat op natuurlijk wijze geincorporeerd worden. De calculus wordt uitgebreid met introductieregels voor procedurele versies van de gewone connectieven voor conjunctie en disjunctie. In de procedurele context hebben deze connectieven een natuurlijke interpretatie: ze implementeren locaal een alternatieve selectieregel. Een beperking van de calculus tot een van deze procedurele connectieven levert een calculus voor een 'snelle' variant op de standaard Prolog computatie, die voor de klasse van links terminerende programma's equivalent is met Prolog.

Voor een correcte reflectie van de effecten van standaard Prolog computatie is een uitbreiding van het Gentzen formaat voor depth first search nodig, die recht doet aan de effecten van Prolog's uitgebreide backtracking. Twee procedurele calculi voor Prolog worden besproken in the vierde hoofdstuk: een met expliciete introductieregels voor een connectief dat Prolog's meest linkse selectieregel reflecteert, en een variant waarin dit connectief in de andere regels geincorporeerd is. In het vijfde hoofdstuk laten we zien hoe de standaard Prolog 'cut' (waarmee de zoekboom gesnoeid kan worden) geincorporeerd kan worden in deze laatste calculus, door toevoeging van twee extra regels. Ook varianten op de Prolog cut worden besproken.

Stellingen

behorende bij het proefschrift

Meta-Logics for Logic Programming

van

Marianne Kalsbeek

1. There exists an Orey-sentence for $\mathrm{I}\Delta_0 + \Omega_1$, that is: There is a sentence G and two interpretations I and J, such that both $\mathrm{I}\Delta_0 + \Omega_1 \vdash (\mathrm{I}\Delta_0 + \Omega_1 + G)^I$ and $\mathrm{I}\Delta_0 + \Omega_1 \vdash (\mathrm{I}\Delta_0 + \Omega_1 + \neg G)^J$.
   Cf. Marianne Kalsbeek, An Orey-sentence for $I\Delta_0 + \Omega_1$, Masters Thesis, ILLC Prepublication Series X-89-01.

2. Modular reasoning in logics with Craig interpolation is complete in the case of (hierarchical) bipartitions, but not in the case of tripartitions.
   Cf. H. Andréka, I. Németi en I. Sain, Craig property of a logic and decomposability of theories, in: Proceedings 9th Amsterdam Colloquium.

3. A logic program P is language independent if and only if the associated Vanilla meta-interpreter V_P is strongly correct w.r.t. P, in the following sense: for all relation symbols R in the language of P and for all terms t, the following holds: $V_P \models solve(R(t)) \Longleftrightarrow P \models R(t)$.

4. The declarative correctness of a *solve*-predicate with respect to terminating object programs can be established without information on the least Herbrand model of such programs. Let P be a terminating program, and let V define a predicate $solve_V$. Then declarative correctness of $solve_V$ with respect to P is equivalent to the following property: for every ground atom in the language of P, $V \models solve(A) \Longleftrightarrow \exists A \longleftarrow B_1, \ldots, B_n \in ground(P)$ s.t. $V \models solve(B_1), \ldots, solve(B_n)$.

5. Gegeven de huidige arbeidsmarkt moet het behalen van de doctorstitel eerder beschouwd worden als afsluiting dan als begin van een wetenschappelijke carrière.