Gentzen systems for Logic Programming styles

Marianne Kalsbeek*
FWI, University of Amsterdam
marianne@fwi.uva.nl

Keywords Logic programming, Prolog, negation as failure, proof theory, cut elimination, selection rule, search rule, substructural logic.

1 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:

Example. The goal A succeeds under SLDNF resolution from the program which consist of the two clauses (1) $A \leftarrow A, B$ (2) $A \leftarrow$. Under Prolog computation however, the search for A will go into a loop.

Example. 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.

The objective of the present paper is to capture the effects of fixed computation mechanisms in the format of Gentzen style sequent calculi. The expressions $[P] \Rightarrow A$ of such calculi will denote the success of the goal A from the program P under the intended computation mechanism.

Typically, computation on a logic program 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 seqent calculi results in so-called 'sub-

*Research supported by the Netherlands Organisation for Scientific Research N.W.O.

structural' 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 $\mathcal{C}$ be a Horn clause. Using top-down clause processing, if a goal A succeeds from the program $P;\mathcal{C};R;Z$ then it also succeeds from $P;\mathcal{C};R;\mathcal{C};Z$, 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 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. The reverse question, "given a proof calculus, is there a sensible computation mechanism to which it corresponds?" will also be addressed. One of the computation mechanisms we successfully axiomatise is standard Prolog computation. To avoid the extra complication of dealing with unification, we will restrict our attention throughout the paper to the propositional case. 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. Another type of extension we will discuss is enrichment of the language. A first extension is addition of negation with the procedural interpretation of negation as finite failure, which can be incorporated in the various calculi in a very natural way. Another possibility we investigate, is to add computational connectives, which can be defined in the standard proof theoretical way, by introduction rules. These connectives have a very natural interpretation in the procedural context: they effect local changes of the overall selection rule.

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. Sometimes we will omit the colon.
- $\emptyset$ 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 suc-

ceeds 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 no $\top$.
- $\mathcal{A}$ 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 setinclusion $P \subseteq R$ for two programs P and R to indicate that every clause of P occurs in R.

Outline of paper.

In the Sections 2–5, we will discuss a computation mechanism characterised by depth-first search and parallel goal processing, and a related Gentzen calculus **S**. We will prove soundness and completeness of calculus with respect to the intended search procedure, using a suitable formal computational semantics. Decidability of the calculus will be proved in Section 5. In Section 6, we will investigate the substructural character of **S** — in particular, we will show that extending **S** with any of the classical structural rules results in a strictly stronger calculus. Several versions of the proof-theoretic cut-rule will be investigated in Section 7.

In Section 8 we will incorporate negation as failure in the calculus **S**. In Section 9, we will show how the underlying language of **S** can be further extended with connectives on goals. These will be incorporated in the calculus by means of proof-theoretic introduction rules. On the level of the computation mechanism, these connectives will be shown to correspond to local changes in the overall selection rule. In Section 10, we focus on one of these procedural connectives, which implements Prolog's leftmost selection rule. In this context, we will discuss a calculus for which we show that the corresponding computation mechanism is essentially a simplification of standard Prolog computation. For a restricted class of programs however, the two computation mechanisms are equivalent.

Finally, in Section 11, we will present a calculus for Prolog computation, and we will prove soundness and correctness of this calculus with respect to Prolog computation. In Section 12, we will discuss some related work. Section 13 is devoted to a short discussion of further issues, open questions and future work.

2 A Gentzen Calculus for Depth-First Search

We will first investigate the following computation mechanism, proposed in [4]:

> **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. The over-all procedure involves backtracking to the last choice point.

The general depth-first search inference mechanism is characterised by Prolog search (depth-first) with a generalised selection rule, to which we will refer 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. [14] and [1]). 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.

We will axiomatise the effects of general depth first search using the format of a Gentzen style sequent calculus, **S**. In this calculus **S**, sequents of the form $[P] \Rightarrow B$ are derivable, where P is a (propositional, Horn clause) program and B is an atom. The intended interpretation is as follows: a sequent $[P] \Rightarrow B$ should be 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

Axiom

$$[P] \Rightarrow \top$$

Heterogeneous Permutation

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

Modus Ponens

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

Prefixing

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

The soundness of the various rules of **S** is almost immediate:

- 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 $A \leftarrow A; P$, 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 an 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 the same as 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. Suppose that a computation of the atom A from P succeeds. 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. 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.

Example 2.1 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 P. 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 $\mathcal{A}$-body atoms succeed in the context of P, and, moreover, they cannot be dependent on A (otherwise, the search for A would get stuck into a loop). Thus all the $\mathcal{A}$-body atoms must succeed in the context of the program in which the uppermost A-clause 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 P if $\mathcal{A}$ is deleted from it—this corresponds 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 can formalize the above soundness and completeness arguments. In order to do this, we will provide **S** with a formal semantics in the next section.

3 A Formal Computational Semantics

In order to formalise the arguments given above for the soundness and completeness of **S**, we will 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:

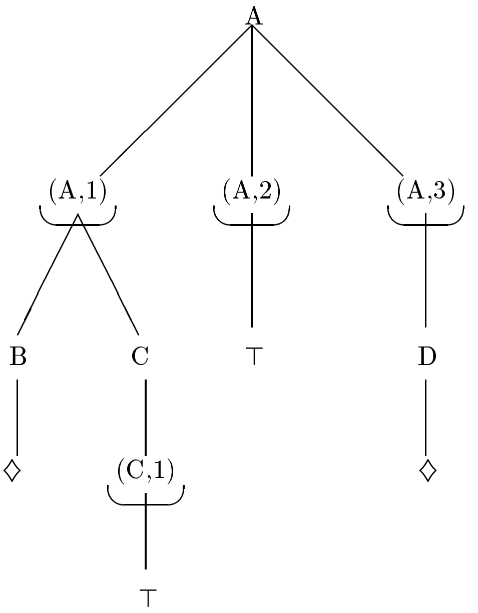
- the top node of the tree is an or-node, labelled with a proper atom, or a terminal node labelled $\top$;
- children of or-nodes are and-nodes, labelled with pairs of proper atoms and natural numbers, or terminal nodes $\diamondsuit$;
- children of and-nodes are or-nodes or terminal nodes;
- terminal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

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

Definition 3.1 An and-or tree T for general depth first search is the *search tree for a goal A from a program P (for A from 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 and there are no B-clauses in P, then it has as single child a terminal node labelled $\diamondsuit$;
- 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. $\Box$

Observe that for every program P and every atom A there is exactly one associated general depth-first search and-or tree.
Below, 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 will consist 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 search-path 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 always 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 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.

Definition 3.2 For atoms A , clauses $\mathcal{C}$, and programs P and R:

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

$\square$

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

Definition 3.3

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 that, for $1 \leq i \leq n$ and $1 \leq j \leq f(i)$, Π_j^i are (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

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

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

5. Every computation is a computation for a goal A from a program P . $\square$

Example 3.1

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.

Lemma 3.1 *There exists a successful computation for A from P iff a search for A from the program P, via general depth-first search, 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.

Definition 3.4 (Subcomputations)

1. Every computation is a subcomputation of itself.

2. If
$$\Pi = A\left[\Pi_1^1 \,|\cdots|\, \Pi_{f(1)}^1\right] \cdots \left[\Pi_1^n \,|\cdots|\, \Pi_{f(n)}^n\right]$$
is a computation, then all subcomputations of Π_j^i, 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.

□

Definition 3.5 (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. □

Definition 3.6 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:

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

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 $P_B = R_B$ for every atom that occurs in Π, then Π is also a computation for A from R.

4 Soundness and Completeness

In the present section, we set out to prove soundness and completeness of the calculus **S** with respect to the computation semantics (Theorem 4.1 and Theorem 4.2). The correctness of **S** for the general depth first search procedure is an immediate consequence of these results and the correctness of the computation semantics with respect to the search procedure (Lemma 3.1).

Theorem 4.1 (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**.)

- AXIOM $\Box$ 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 insensitive to heterogeneous permutation in programs.

- 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 $\mathbf{S}$. By inductive hypothesis there is a successful computation $A[\sigma_1] \cdots [\sigma_m]$ for A from P for some m. Also, there are successful computations Π_i for B_i from $A \leftarrow A; P; B_i \leftarrow$. By 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 P_i. If all the Π_i^* are successful computations, then (by definition) $A[\Pi_1^* | \cdots | \Pi_n^*]$ is a successful computation for A from $A \leftarrow A; P; B_i \leftarrow$. Otherwise, again by definition, $A[\Pi_1^* | \cdots | \Pi_n^*][\sigma_1 \cdots [\sigma_m]$ is a successful computation for A from $A \leftarrow B_1, \ldots, B_n; P$. $\Box$

Theorem 4.2 (Completeness) *If Π is a successful computation for A from P, then $\mathbf{S} \vdash [P] \Rightarrow A$.*

PROOF.
The only successful computation of depth 1, $\Box$, 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 some A from P, and $\mathtt{depth}(\Pi) = k$.
We distinguish two cases.

- $n = 1$, that is, $\Pi = A \left[\Pi_1 | \cdots | \Pi_{f(1)}\right]$.
  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 the sequent $\left[A \leftarrow B_1, \ldots, B_n; P^{-A}\right] \Rightarrow A$. 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 $t < n$. First observe that the case $t = n - 1$ is already treated above.
  Suppose that for some $s > 0$, $\left[P^{s,A}\right] \Rightarrow A$ is derivable. Now, by definition, the P_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 propery, 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 the prefixing rule 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. □

5 Decidability

In the present section, we attend to some classical proof-theoretic issues in the context of **S**, in particular decidability (Section 5.2) and the existence of normal forms for derivations (Section 5.1). Decidability of **S** could be easily inferrend 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.

5.1 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 derviations consist of axioms followed by one application of MP, and applications of HP.

(1)

$$\dfrac{\begin{array}{c}\vdots\\ [A \leftarrow A; A \leftarrow B; C; B] \Rightarrow C\end{array}}{[A \leftarrow C; A \leftarrow B; C; B] \Rightarrow A}\,\mathrm{MP}$$

(2)

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ [A \leftarrow A; C; B] \Rightarrow B\end{array}}{[A \leftarrow B; C; B] \Rightarrow A}\,\mathrm{MP} \qquad \begin{array}{c}\vdots\\ [A \leftarrow A; A \leftarrow B; C; B; C] \Rightarrow C\end{array}}{[A \leftarrow C; A \leftarrow B; C; B] \Rightarrow A}\,\mathrm{PFX}$$

The difference between these two different derivations can be thought of as follows. In the uppermost dervation (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 latter 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:

Proposition 5.1 *Every* $\mathbf{S}$*-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.*

5.2 Decidability

We will prove decidability of $\mathbf{S}$ 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 conditionto these rules. This follows from the following proposition.

Proposition 5.2 *If* $[P] \Rightarrow A$ *is derivable in* $\mathbf{S}$*, 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,

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

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,

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

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. $\square$

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:

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

Then, by the above, the modified calculus, say **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:

Proposition 5.3 *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. 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 of a clause $\mathcal{A}$ as follows:

$v(\mathcal{A}) = 0$ iff $\mathcal{A}$ is a reflexive clause $A \leftarrow A$ or a fact-clause $A \leftarrow$,
$v(\mathcal{A}) = 1$ 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 of sequents $[P] \Rightarrow A$ as follows:

$$v([P] \Rightarrow \top) = 0,$$
$$v([P] \Rightarrow A) = v(P) + 1 \qquad \text{if } A \neq \top.$$

For any instance of the MP and PFX rules in **S**', the weight of the assumptions is strictly smaller than the weight of the conclusion. (In contrast, this does not hold for **S**.) As a result of the above observations, we have the following.

Proposition 5.4 *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 $v([P] \Rightarrow A)$.*

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.

Proposition 5.5 *Let P be a program $P = \mathcal{C}_1, \ldots \mathcal{C}_n$, and let k be minimal such that $\mathcal{C}_k$ is an A-clause. Then $[P] \Rightarrow A$ is derivable if and only if the sequent $[\mathcal{C}_k, \mathcal{C}_1, \ldots, \mathcal{C}_{k-1}, \mathcal{C}_{k+1}, \ldots, \mathcal{C}_n] \Rightarrow A$*
(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 dervation 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 $\mathcal{C}_1, \ldots, \mathcal{C}_n$, 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 5.4 and 5.5, 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 $2n$. 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.

We can conclude that

Theorem 5.6 **S** *is decidable.*

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

6 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 distingiush 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 derivablility. In the latter case, we call R *sound* with respect to C.

In Figure 1, we list the proper formulations of the classical structural rules (Exchange EX, Contraction, and Weakening W) and the usual cut rule (Classical

Exchange

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

Weakening

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

Contraction

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

Classical Cut

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

Figure 1: Classical rules

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 results in a strictly stronger calculus (Section 6.2 and 7).

In Section 6.1 and 7 we will discuss several weak versions of these classical rules, listed in Figure 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 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. [9]).

Definition 6.1
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:

Proposition 6.1 *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 (that is, for all sequents*

Rightward Extension

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

Atomic Monotonicity

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

Full Context Cut

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

Partial Context Cut

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

Figure 2: Substructural rules

s, if $C + R \vdash s$, then $C \vdash s$). 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.

Proposition 6.2 *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 to the above, admissibility does not imply derivablility. In Section 6.1 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:

Definition 6.2 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 $T \preceq S$.

6.1 Admissible rules for S

In [4], 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 will show that these rules are indeed admissible in **S** (Proposition 6.3 and 6.4). 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.

Proposition 6.3 *Rightward Contraction is admissible in* **S**.

PROOF. We first show that the following holds.

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 $Q;\mathcal{B};R;Z$.

Let, in the above, $\Pi = B[\delta_1]\cdots[\delta_n]$. Also, let the leftmost and rightmost exhibited $\mathcal{B}$ 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 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 successful (failing) computation for B from $Q;\mathcal{B};R;Z$.
3. $n = k$. Then Π is a failing computation, and $\Pi' = B[\delta_1]\cdots[\delta_{k-1}]$ is the failing computation for B from $Q;\mathcal{B};R;Z$.

Now let Σ be the successful computation for A from $Q;\mathcal{B};R;\mathcal{B};Z$. Suppose that B does not occur in Σ. Then, by the inessential difference property, Σ is also the successful computation for A from $Q;\mathcal{B};R;Z$. Suppose that B does occur in Σ. Replacing all occurrences of Π in Σ by Π' results in a successful computation for A from $Q;\mathcal{B};R;Z$. By soundness and completeness of **S** with respect to computations, the proposition follows. $\square$

By similar arguments, we have the following.

Proposition 6.4 *Rightward Extension (RE) is admissible in* **S**.

In contrast, the following holds.

Proposition 6.5 *Neither Rightward Extension nor Rightward Contraction are derivable rules in* **S**.

PROOF. By Proposition 6.2, it suffices to find rule extensions of **S** in which Rightward Extension and Rightward Contraction are not admissible. In particular, by Proposition 6.1 it suffices to find rules R_E and R_C and sequents T_E and T_C such that

1. $\mathbf{S}$ + R_E + RE $\vdash T_E$, while $\mathbf{S}$ + $\mathrm{R}_E \not\vdash T_E$;
2. $\mathbf{S}$ + R_C + 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:

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

and let T_C be the sequent $[A \leftarrow A;\ A \leftarrow] \Rightarrow A$. Now T_C and R_C are as in (2). We have the following derivation of T_C in $\mathbf{S}$+ R_C + RC:

$$\cfrac{\cfrac{\cfrac{\cfrac{[A \leftarrow A; A \leftarrow A; A \leftarrow] \Rightarrow \top}{[A \leftarrow; A \leftarrow A; A \leftarrow] \Rightarrow A}\,\mathrm{MP}}{[A \leftarrow A; A \leftarrow; A \leftarrow A; A \leftarrow] \Rightarrow A}\,\mathrm{R}_C}{[A \leftarrow A; A \leftarrow; A \leftarrow] \Rightarrow A}\,\mathrm{RC}}{[A \leftarrow A; A \leftarrow] \Rightarrow A}\,\mathrm{RC}$$

The arguments used above in case 1) also apply here, to show that $\mathbf{S}$+ $\mathrm{R}_C \not\vdash T_C$. This concludes the proof of 2). $\square$

Another sound but not 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 that 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.

6.2 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:

Theorem 6.6

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

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 for Horn clauses. 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**:

Theorem 6.7 $\mathbf{S} + \mathrm{EX} \equiv \mathbf{S} + \mathrm{LC} + \mathrm{RC} \equiv \mathbf{S} + \mathrm{W}$.

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 successful 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 of Π, then $\Sigma = \Gamma$.

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 up to permutation.

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 LC 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 and Γ are subcomputations of Π_i and Π_j, respectively, for C, Σ is identical to Γ. 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 $[A] \Rightarrow B_1, \ldots, B_n; P$.

Completeness
By induction on the (natural) depth of bf-computations. Suppose that for all atoms A, programs P, and bf-computations for A from P of depth $< n$, the sequent $[P] \Rightarrow A$ is derivable from any of the relevant calculi. Let $A[\Pi_1|\cdots|\Pi_n]$ 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 Qoccurrence 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 δ_i, γ_i, and ρ_i, in respectively $\mathbf{S}$ + W, $\mathbf{S}$ + EX, and $\mathbf{S}$ + LC + RC. Indicating consecutive applications of one and the same rule R with a star *, we find the following derivations of the sequent $[P] \Rightarrow A$.

$$\cfrac{\cfrac{\cfrac{\overset{\delta_1}{[A \leftarrow A; Q'] \Rightarrow B_1} \qquad \cdots \qquad \overset{\delta_n}{[A \leftarrow A; Q'] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; Q'] \Rightarrow A}\,\mathrm{MP}}{[Q] \Rightarrow A}\,\mathrm{HP}^*}{[P] \Rightarrow A}\,\mathrm{W}^*$$

$$\cfrac{\cfrac{\overset{\gamma_1}{[A \leftarrow A; P'] \Rightarrow B_1} \qquad \cdots \qquad \overset{\gamma_n}{[A \leftarrow A; P'] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; P'] \Rightarrow A}\,\mathrm{MP}}{[P] \Rightarrow A}\,\mathrm{EX}^*$$

$$\cfrac{\cfrac{\overset{\rho_1}{[A \leftarrow A; P] \Rightarrow B_1} \qquad \cdots \qquad \overset{\rho_n}{[A \leftarrow A; P] \Rightarrow B_n}}{[A \leftarrow B_1, \ldots, B_n; P] \Rightarrow A}\,\mathrm{MP}}{[P] \Rightarrow A}\,\mathrm{LC}$$

This completes the completeness proof. Observe that the above shows that the PFX rule is redundant in any of the three relevant extensions. In addition, Q' can be chosen in such a way that applications of HP are not necessary — thus, HP is redundant is well. Also, RC is redundant in $\mathbf{S}$+ LC + RC. □

Observe also that $\mathbf{S}$ + EX, as it corresponds to breadth first search, is in fact equivalent to classical logic CL for Horn clause theories. We thus find ourselves in the following substructural landscape:

$$\begin{array}{c} \mathrm{CL} \equiv \mathbf{S} + \mathrm{EX} \equiv \mathbf{S} + \mathrm{LC} + \mathrm{RC} \equiv \mathbf{S} + \mathrm{W} \\ | \\ \mathbf{S} \end{array}$$

7 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):

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

(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 Theorem 6.7, shows that **S** + Classical Cut is intermediary between **S** and any of the extensions with Weakening, Contraction, or Exchange.

Theorem 7.1 **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, first observe that

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

In contrast, we will show that
$$\mathbf{S} + \text{Classical Cut} \nvdash [p \leftarrow p; p \leftarrow] \Rightarrow p. \tag{b}$$

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

[1] 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.

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 which occur in P_n are $p \leftarrow$ and $p \leftarrow p$.

Take for $P_0 = p \leftarrow p; p \leftarrow$. Then $[P_0] \Rightarrow p$ is the conclusion of d, and (1) and (2) hold for P_0 . Suppose (1) and (2) hold for P_{n-1}. Then P_{n-1} cannot be and 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 take $P_n = P_{n-1}$.
- 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:

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

  where $P_{n-1} = T; Y; Z$.
  Because by inductive hypothesis (2) holds for P_{n-1}, 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:

CL ≡ **S** + EX ≡ **S** + LC + RC ≡ **S** + W

|

S + CC

|

S

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**.

Theorem 7.2 *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 Σ for all subcomputations of Π for C from $P; C; R$ results in a (successful) computation for A from $P; R$. (As witnessed by the above counterexample, a similar substitution 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 $P; Q; R$.) $\square$

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:

Theorem 7.3 *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, Π is also the successful computation for $Q; R; Z$. So suppose C occurs 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. $\square$

Corollary 7.1 *There exists an effective procedure for the elimination of Partial Context Cut.*

Comment. Several questions can be raised at this stage. Are Partial and Full context Cut derivable rules for **S**? Are there other admissible cut rules for **S** besides Partial and Full Context Cut? Also, shifting our point of view to the procedural level, the status of the various cut rules is unclear. If we regard the context cut rules as program transformation rules, their practicality 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. It is also an open question whether there is a sensible proof procedure that corresponds to the system **S** + Classical Cut. An interesting issue not addressed here is the so-called rule completeness for **S**: an effective characterisation of the derivable and the admissible rules.

8 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 $\mathbf{S}_\neg$.

8.1 Notation and conventions

- L is a literal if L is either A or $\neg A$, where A is an atom.
- L and K (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:

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

- $\mathcal{A}$ 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' $\sum_{i=1}^{n} [P_i] \Rightarrow E_i$ to indicate the list of n assumptions $[P_1] \Rightarrow E_1$, ..., $[P_n] \Rightarrow E_n$.

8.2 Axiomatising negation as failure

We axiomatise depth first search with negation as failure in a deductive calculus $\mathbf{S}_\neg$, consisting of the following set of axioms and rules:

$$\mathbf{S}_\neg$$

Axioms

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

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

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

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

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

The soundness of $\mathbf{S}_\neg$ is almost immediate:
Clearly, the axiom 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 and the prefixing rule PFX, both formulated for normal clauses, are sound. These rules and axioms deal with success of atoms.

The new axiom AX$_\neg$ and the new rule PFX$_\neg$ deal with the success of negative literals (i.e., by the duality principle, with the failure of atoms). AX$_\neg$ expresses the immediate failure of undefined atoms. 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.

8.3 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 $D[E[\diamondsuit]]$. So $\neg D$ succeeds from P, and the associated successful computation is $\neg D[E[\diamondsuit]]$.
The successful computation for A from P is $A[B[C[\Box]]|\neg D[E[\diamondsuit]]]$, 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 $\mathbf{S}_\neg$.

Theorem 8.1 *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)

PROOF. 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 $\mathbf{S}$. 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 follows from the fact that the semantical objects formalise search through the relevant search trees. □

9 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 [16]. We will, in the present context, not pursue this possibility.

3. At another level, connectives between clauses and on programs themselves. 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. [6] and [7]). 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.

9.1 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 both the success conditions and 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 *directed choice* A/B, which works as follows. Start with a search for A. Success for A already gives success of A/B, while failure induces a further search for B. Success of B then gives success for A/B, while 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 possible 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		$A \cap B := \neg(\neg A \| \neg B)$
DIRECTED				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. [20]), 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 before would, in the present context, require extended literals to occur in the head of (fact-)clauses. Therefore, $\mathbf{S}^*$ will use the following alternative formulation of the prefixing rules.

Prefixing PFXN

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

where $1 \leq k \leq n$.

Prefixing PFX$_\neg$

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

where $1 \leq k \leq n$.

In addition to the above mentioned rules, $\mathbf{S}^*$ has the usual 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:

Proposition 9.1 *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 9.1) and the calculus $\mathbf{S}^*$ is as expected:

Proposition 9.2 *For all extended literals L and all programs P with only atoms as heads, $\mathbf{S}^* \vdash P \Rightarrow L$ iff there exists a successful computation for L from P.*

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

9.3 Connectives and selection rules

There is a close connection between the 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. 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 $\mathbf{S}$-conjunctions with juxtaposition comma's. (For simplicity, assume that the $\mathbf{S}$-conjunctions do not occur under any other connective than S-conjunctions.)

Now let C be a succeeding (failing) $\mathbf{S}^*$-computation for L from P. A corresponding succeeding (failing) computation C' for L from P' can be obtained by an obvious transformation of C (replace all subcomputations of the form $(K \rhd L)P{\star}Q$ by juxtapositions $K\,[P]\,|L\,[P]$. 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, 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 directed composition in the context of $\mathbf{S}^*$ and Prolog in the next sections.

10 Frugal Prolog

In this section we take a closer look at directed composition. As we have observed above in Section 9.3, directed composition $\angle$ corresponds to leftmost goal selection. That is, in the context of $\mathbf{S}^*$, directed composition locally implements Polog'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.

10.1 Frugal Prolog

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

Definition 10.1 (generalised directed composition)

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

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, 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

Proposition 10.1 *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 by standard Prolog, by the use of commit operators on goals or by the use of the standard Prolog cut !.

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 resonable alternative for standard Prolog. In addition, frugal Prolog travels through the search space faster than standard Prolog. These observations justify a closer look at frugal Prolog.

10.2 Characterising frugal Prolog

We can do much better than the implicit characterisation of frugal Prolog in the context of $\mathbf{S}^*$. We give a calculus $\mathbf{S}_{fPr}$ which correctly axiomatises frugal Prolog, and which, unlike $\mathbf{S}^*$, only derives sequents in Prolog form. Instead of introducing directed composition by proof theoretic introduction rules, we use the procedural interpretation of directed composition to define appropriate Modus Ponens and Prefixing rules. As connectives on goals we will only admit generalised directed composition and negation ($\neg$).

$$\mathbf{S}_{fPr}$$

AX

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

$\mathrm{AX}_\neg$

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

Modus Ponens

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

Prefixing

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

where $L^+ = A$ and $1 \leq k \leq n$

We have the following correctness theorem.

Theorem 10.2 *Let $[P] \Rightarrow L$ be in Prolog form. Then $\mathbf{S}^* \vdash [P] \Rightarrow L$ iff $\mathbf{S}_{fPr} \vdash [P] \Rightarrow L$.*

Proof. From right to left. This is in fact a direct consequence of the considerations that led to the definition of $\mathbf{S}_{fPr}$. Observe that every instance of the axioms of $\mathbf{S}_{fPr}$ is an instance of an axiom of $\mathbf{S}^*$. Every instance of MP in $\mathbf{S}_{fPr}$ is derivable in $\mathbf{S}^*$, by n-1 consecutive applications of the introduction rule for directed composition. Every instance of PFX in $\mathbf{S}_{fPr}$ is derivable in

$\mathbf{S}^*$ by $k-1$ consecutive applications of the introduction rule for the negation of directed composition followed by an application of PFXN or $\text{PFX}_{\neg}$.

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 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:

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

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

The prefixing rules of $\mathbf{S}^*$ are analogous. $\square$

A a direct consequence of Proposition 10.1 and Theorem 10.2, we have the following Theorem.

Theorem 10.3 *Let $[P] \Rightarrow L$ be a sequent in Prolog form. Then L succeeds from P via the frugal Prolog computation mechanism iff $\mathbf{S}_{fPr} \vdash [P] \Rightarrow L$.*

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, the two computation mechanisms are equivalent. Indeed, this is true for the class of *left terminating* (or acceptable) programs [2]. A (propositional) program P is left terminating if all LDNF-derivations on P are finite.

Theorem 10.4 *Let P be a left terminating program. Then $\mathbf{S}_{fPr} \vdash [P] \Rightarrow L$ iff L Prolog-succeeds on P.*

Proof. Let P be left terminating. Then every literal succeeds or finitely fails on P via standard Prolog search, as every LDNF-derivation from P is finite. The same is true for 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 $P \cup \{L\}$. $\Box$

11 Prolog

None of the calculi discussed thus far properly reflects standard Prolog computation with leftmost goal selection and depth-first computation. The above results and counterexamples suggest that in order to obtain a procedural calculus which is both sound and complete 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 a sequent calculus ($\mathbf{S}_{Pr}$), for which we will show soundness and completeness with respect to the standard Prolog computation mechanism. This calculus for Prolog is defined in an extension of the Gentzen style sequent calculus format we used thus far. The extension consists in an addition, to the basic language of Gentzen style sequent calculus, of an alternative implication $\Rightarrow^*$. Thus we work in a sequent calculus format with two kinds of basic expressions: the usual sequents $[P] \Rightarrow A$, and new (star-) sequents $[P] \Rightarrow^* A$. While the usual 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 LD-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.

The idea of using a 'two-sorted' calculus in the present setting is not original. In [4], the use of a second implication $\Rightarrow^-$ was suggested to express finite failure of the succedent in absence of negation in the language. (Another example of the use of 'sorted' calculi are second order calculi, with implications between sequents.)

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 $\neg$ 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.

11.1 A calculus for Prolog

The calculus for Prolog, $\mathbf{S}_{Pr}$, of which the complete list of rules can be found in the Figures 3 and 4, contains axioms expressing the immediate success of the empty goal $\top$ and the immediate failure of undefined atoms. As usual, the HP rule expresses Prolog's insensitivity to the relative order of clauses with different heads.

For a motivation of the remaining rules, which deal with the success and finite failure of *defined* atoms A from a program R, we distinguish several cases. It will be useful to assume (without loss of generality, by heterogeneous permutation) that

$$R = A \leftarrow B_1, \ldots, B_n; P.$$

Thus, we can split the LD-tree $\mathcal{T}$ for $R \cup \{A\}$ in two parts $\mathcal{T}_L$ and $\mathcal{T}_R$ where $\mathcal{T}_L$ is the subtree of $\mathcal{T}$ which is the LD-tree for $R \cup \{B_1, \ldots, B_n\}$, and where $\mathcal{T}_L$ is the remainder of $\mathcal{T}$, which roughly corresponds to the remaining A-clauses in P. A Prolog search for A from R will first enter $\mathcal{T}_L$, and end in success if $\mathcal{T}_L$ is successful. If $\mathcal{T}_L$ is finitely failed, the search will subsequently enter $\mathcal{T}_R$.

We distinguish two cases.

1. The atom A succeeds on the program R. (That is, $\mathcal{T}$ is successful.)
   As before, we can distinguish two cases.

   (a) A succeeds via the leftmost A-clause in R. (That is, $\mathcal{T}_L$ is successful.) This corresponds to the usual modus ponens rule MP.

   (b) A fails via the leftmost A-clause $A \leftarrow B_1, \ldots, B_n$, but succeeds via some further A-clause in P. (That is, $\mathcal{T}_L$ is finitely failed, while success occurs in $\mathcal{T}_R$.)

   As we have observed in the previous section, the usual prefixing rule does not properly reflect this case, due to Prolog's extensive backtracking. By the leftmost selection rule, failure of some B_m (with $m \leq n$) and success of the preceding $B_1, \ldots, B_{m-1}$ is a necessary condition. That it is not sufficient is caused by the Prolog's backtracking. After establishing the failure of B_m, the system backtracks and goes into a renewed search for the composite goal B_{m-1}, B_m via yet untried clauses with head B_{m-1}. Divergence of this subsearch would contradict the eventual success of A. Also, as B_m finitely fails, the search for B_{m-1}, B_m will eventually, after repetition of the backtracking on B_{m-1}, fail finitely. Then the system will backtrack again and go into a renewed search for the composite goal B_{m-2}, B_{m-1}, B_m, via yet untried clauses with head B_{m-2}. This procedure is continued until all possible clauses for $B_1, \ldots, B_m$ are tried. This extensive backtracking generates a complete lefmost, depth-first search through the search tree for $B_1, \ldots, B_m$. Therefore, usual prefixing conditions have to be strengthened in such a way that termination of the search, or, equivalently, finiteness of the LD-tree for $B_1, \ldots, B_m$, is ensured. Now observe that finiteness of the LD tree for $B_1, \ldots, B_m$, under condition of finite failure of B_m, and success of $B_1, \ldots, B_{m-1}$, is equivalent to finiteness of the respective LD-trees for $B_1, \ldots, B_{m-1}$. This is where we use stronger star-sequents, to express success in combination with finiteness of the LD-tree. The resulting prefixing rule is PFX_1.

(Obviously, we also need rules to derive star-sequents. These rules will be discussed below.)

2. The atom A fails finitely from the program R. That is, A fails via the leftmost A-clause $A \leftarrow B_1, \ldots, B_n$, 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 previous case 1(b), to the modified prefixing assumptions for the failure of a composite goal. The resulting prefixing rule is PFX_2.

Furthermore, we need rules and axioms to derive star-sequents. Keeping in mind the intended meaning of the star-sequents (success and finiteness of the relevant LD-tree), we obviously have 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 motivation above for the various rules which have as a consequent a non-star sequent $[R] \Rightarrow A$, where A is defined in R, we can distinguish several cases. Relevant for the 'star-rules' is that the finiteness of the LD-tree $\mathcal{T}$ is equivalent the finiteness of both $\mathcal{T}_L$ and $\mathcal{T}_R$. We can distinguish the following two cases here:

1. A succeeds on the leftmost A-clause, that is, $\mathcal{T}_L$ is successful as well as finite.
   This case corresponds to a strengthening of the modus ponens rule. That is, $\mathcal{T}_L$ is successful, and both $\mathcal{T}_L$ and $\mathcal{T}_R$ are finite. Success and finiteness of $\mathcal{T}_L$ translates into star versions of the usual assumptions for modus ponens. Furthermore, two cases can be distinguished: either $\mathcal{T}_R$ contains a successbranch or all its branches are failed. The first case corresponds to success of A from P (with a finite LD-tree), and gives rise to an extra assumption $[P] \Rightarrow^* A$. This results in the rule MP*. The second case corresponds to finite failure of A from P, and gives rise to an extra assumption $[P] \Rightarrow \neg A$. This results in the rule MP#.

2. A fails via the leftmost A-clause $A \leftarrow B_1, \ldots, B_n$, but succeeds via some further A-clause in P, while $\mathcal{T}_R$ is finite. The first condition translates into the appropriate assumptions expressing failure of a composite goal, as in the case 1(b). The second condition translates into the assumption $[P] \Rightarrow^* A$. This gives us 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 soundness and completeness 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 computations.

Proposition 11.1
For all atoms A, if $\mathbf{S}_{Pr} \vdash [P] \Rightarrow^* A$ then $\mathbf{S}_{Pr} \vdash [P] \Rightarrow A$.

Modulo the correctness of $\mathbf{S}_{Pr}$ with respect to Prolog computation, it is now simple to establish the *completeness* of the frugal Prolog computation mechanism with respect to Prolog computation. This can be shown by induction on $\mathbf{S}_{Pr}$- derivations, and using Proposition 11.1.

Theorem 11.2 *If $\mathbf{S}_{Pr} \vdash [P] \Rightarrow L$ then $\mathbf{S}_{fPr} \vdash [P] \Rightarrow L$.*

As we have observed in the previous section, $\mathbf{S}_{fPr}$ is not *sound* with respect to Prolog computation. That is, the converse of Theorem 11.2 does not hold. This can be illustrated by the program $A \leftarrow$; $A \leftarrow A$; $B \leftarrow A, C$; $B \leftarrow$. The goal B will succeed from this program via frugal Prolog computation, while a standard Prolog computation will diverge while trying to solve A via the reflective A-clause $A \leftarrow A$.

11.2 Soundness and completeness

We remind the reader of some relevant notation and concepts, and introduce some new ones.

- An LD-tree for $P \cup \{L_1, \ldots, L_n\}$ is an SLD-tree for $P \cup \{L_1, \ldots, L_n\}$ via the leftmost selection rule. The LD-tree for $P \cup \{L_1, \ldots, L_n\}$ is unique.
- For leaves C in $\mathcal{T}$, there are two possibilities:
  - $C = \Box$. Then C is marked as success.
  - 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 *successbranch* if it ends in a leaf marked as success.
- An LD-tree $\mathcal{T}$ is successful for depth-first search, or in short *successful*, if it has a successbranch such that all branches left of it end in a leaf marked as failure.
- An LD-tree is *finitely failed* if ($\mathcal{T}$ is finite and) all its branches end in a leaf marked as failure.

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

$\mathbf{S}_{Pr}$

Axioms

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

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

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

HP/HP$*$

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

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

MP

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

MP$*$

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A;P] \Rightarrow^* B_1 \qquad \ldots \qquad [A \leftarrow A;P] \Rightarrow^* B_n}{[A \leftarrow B_1,\ldots,B_n;P] \Rightarrow^* A}$$

MP$\#$

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

Figure 3: Axioms, Heterogeneous Permutation, Modus Ponens

PFX$_1$

$$\frac{[P] \Rightarrow A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \quad \ldots \quad [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg B}{[A \leftarrow B_1, \ldots, B_n, B, C_1, \ldots, C_m; P] \Rightarrow A}$$

PFX$_2$

$$\frac{[P] \Rightarrow \neg A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \quad \ldots \quad [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg B}{[A \leftarrow B_1, \ldots, B_n, B, C_1, \ldots, C_m; P] \Rightarrow \neg A}$$

PFX$_3$

$$\frac{[P] \Rightarrow^* A \quad [A \leftarrow A; P] \Rightarrow^* B_1 \quad \ldots \quad [A \leftarrow A; P] \Rightarrow^* B_n \quad [A \leftarrow A; P] \Rightarrow \neg B}{[A \leftarrow B_1, \ldots, B_n, B, C_1, \ldots, C_m; P] \Rightarrow^* A}$$

Figure 4: Prefixing rules of $\mathbf{S}_{Pr}$

Lemma 11.3
(a) The LD-tree for $P \cup \{L_1, \ldots, L_n\}$ is successful iff the goal $L_1, \ldots, L_n$ succeeds from P via standard Prolog computation.
(b) The LD-tree for $P \cup \{L_1, \ldots, L_n\}$ is finitely failed iff the goal $L_1, \ldots, L_n$ finitely fails from P via standard Prolog computation.

Of every LD-tree $\mathcal{T}$, we can take 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.

Definition 11.1 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 successbranch and all (finitely 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 (finitely failed) branches left of it. $\square$

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.

Definition 11.2 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:

Definition 11.3 $\langle a, b \rangle <_o \langle c, d \rangle$ 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 an LD-trees for a composite goal, and the LD-trees for the components. We omit the proofs.

Lemma 11.4 *Let $\mathcal{T}$ be the LD-tree for $P \cup \{L_1, \ldots, L_n\}$, and let for $i \in [1, n]$, $\mathcal{T}_i$ be the LD-tree for $P \cup \{L_i\}$. Then*
(a) If $\mathcal{T}$ is successful, then, for $i \in [1, n]$, $\mathcal{T}_i$ is successful and $w(\mathcal{T}_i) \leq_o w(\mathcal{T})$.
(b) If, for $i \in [1, n]$, $\mathcal{T}_i$ is successful, then $\mathcal{T}$ is successful.

Lemma 11.5 *Under the conditions of Lemma 11.4, 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 $w(\mathcal{T}_i) \leq_o w(\mathcal{T})$.
(b) If $\mathcal{T}_i$ is successful and finite, for $i \in [1, n]$, then $\mathcal{T}$ is successful and finite.

Lemma 11.6 *Under the assumptions of Lemma 11.4, 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 $w(\mathcal{T}_i) \leq_o w(\mathcal{T})$, for $i \leq k_{\mathcal{T}}$.

The following lemmas are the proper equivalents of the inessential difference and occurrence properties of formal computations.

Lemma 11.7 *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 searchpath of $\mathcal{T}$, then the searchpath of $\mathcal{T}$ is equal to the searchpath of $\mathcal{S}$.*

3. *If A is never selected in $\mathcal{T}$ then $\mathcal{S}$ is equal to $\mathcal{T}$.*

Lemma 11.8 *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 searchpath of $\mathcal{T}$.*

Lemma 11.9 *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 searchpath of $\mathcal{T}$, except in the root of $\mathcal{T}$.*

We are now in position to state and prove the main theorem of this section.

Theorem 11.10
$\mathbf{S}_{Pr} \vdash [P] \Rightarrow A$ *iff L succeeds from A via standard Prolog computation.*
$\mathbf{S}_{Pr} \vdash [P] \Rightarrow \neg A$ *iff A finitely fails from P via standard Prolog computation.*

Proof. We prove the following claim. The theorem immediately follows from the claim and Lemma 11.3.

Claim.

1. $\mathbf{S}_{Pr} \vdash [P] \Rightarrow A$ *iff the LD-tree for $P \cup \{A\}$ is successful.*

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

3. $\mathbf{S}_{Pr} \vdash [P] \Rightarrow \neg A$ *iff the LD-tree for $P \cup \{A\}$ is finitely failed.*

Proof of the claim. 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 consisting only of the application of an axiom, 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.

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

$$\dfrac{\begin{matrix}\vdots & & \vdots \\ [A \leftarrow A; P] \Rightarrow B_1 & \dots & [A \leftarrow A; P] \Rightarrow B_n\end{matrix}}{[A \leftarrow B_1, \dots, B_n; P] \Rightarrow A}\ \mathrm{MP}$$

By the inductive hypothesis, the LD-trees for $A \leftarrow A\,;\, P \cup \{B_i\}$ are successful. Using the Lemmas 11.4(b), 11.8(2), and 11.7(2), we infer that the LD-tree for $A \leftarrow B_1, \dots, B_n\,;\, P \cup \{B_1, \dots, B_n\}$ is successful. Thus the LD-tree for $A \leftarrow B_1, \dots, B_n; P \cup \{A\}$ is successful.

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

$$\dfrac{\begin{matrix}\vdots & \vdots & & \vdots \\ [P] \Rightarrow^* \neg A & [A \leftarrow A; P] \Rightarrow^* B_1 & \dots & [A \leftarrow A; P] \Rightarrow^* B_n\end{matrix}}{[A \leftarrow B_1, \dots, B_n\,;\, P] \Rightarrow^* A}\ \mathrm{MP}^*$$

Let $\mathcal{T}$ be the LD-tree for $A \leftarrow B_1, \dots, B_n\,;\, P \cup \{A\}$. We will show that $\mathcal{T}$ is successful and finite. Let $\mathcal{L}$ be the LD-tree for $A \leftarrow B_1, \dots, B_n\,;\, P \cup \{B_1, \dots, B_n\}$, and let $\mathcal{R}$ be the LD-tree for $P \cup \{A\}$. By the inductive hypothesis, $\mathcal{R}$ is finitely failed. Also by the inductive hypothesis, the LD-trees for $A \leftarrow A\,;\, P \cup \{B_i\}$ are finite and successful. Thus, from the Lemmas 11.5(b), 11.8, and 11.7(3), it follows that $\mathcal{L}$ is successful and finite. Therefore, $\mathcal{T}$ is successful. 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 $b_2 = B_1, \dots, B_n$. Then $b_2, b_3, \ldots$ is a branch in $\mathcal{L}$. Thus b is finite. So suppose $b_2 \neq B_1, \dots, B_n$. If A is never selected on $b_2, b_3, \ldots$, then b is (by Lemma 11.7(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 $b_1, \dots, b_k$ 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 11.9(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:

$$\frac{\begin{matrix}\vdots & \vdots & & \vdots & \vdots\\ [P] \Rightarrow A & [A \leftarrow A; P] \Rightarrow^* B_1 & \dots & [A \leftarrow A; P] \Rightarrow^* B_n & [A \leftarrow A; P] \Rightarrow \neg B\end{matrix}}{[A \leftarrow B_1, \dots, B_n, B, C_1, \dots, C_t\, ;\, P] \Rightarrow A}$$

By the inductive hypothesis and the Lemmas 11.6, 11.8(1), and 11.7(3), the LD-tree for $A \leftarrow B_1, \dots, B_n, B, C_1, \dots, C_t; P \cup \{B_1, \dots, B_n, B, C_1, \dots, C_t\}$ is finitely failed. By inductive hypothesis the LD-tree for $P \cup \{A\}$ is successful. The Lemmas 11.9(2) and 11.7 can be applied to show that the LD-tree for $A \leftarrow B_1, \dots, B_n, B, C_1, \dots, C_t; P \cup \{A\}$ 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 failing LD-tree is finite, the number of nodes in a successful tree might be infinite. As a result, we need tranfinite 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 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 $P = A \leftarrow B_1, \dots, B_n; R$ for some R. Thus, the root A of $\mathcal{T}$ has for its leftmost child the node $B_1, \dots, B_n$.

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 $P \cup \{B_1, \dots, B_n\}$. 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 $A \leftarrow A\, ; R \cup \{B_1, \dots, B_n\}$;

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

We will use the following observations:

- By assumption, $\mathcal{T}$ is successful or finitely failed. Thus, by Lemma 11.9, A is never selected on the search path of $\mathcal{T}$. The search path of $\mathcal{T}$ extends the search path of $\mathcal{T}_L$. Consequently, A is never selected on the search path of $\mathcal{T}_L$. Thus, by Lemma 11.7, 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 11.9(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 11.7,

  (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:

  (6) $w(\mathcal{R}) <_o w(\mathcal{T})$
  (7) $w(\mathcal{L}) <_o w(\mathcal{T})$.

- Suppose $\mathcal{T}$ is not finite. We distinguish the following cases:

  - $\mathcal{T}_L$ is finitely failed. Then, by Lemma 11.7, $\mathcal{T}_L$ is identical with $\mathcal{L}$. Thus $\sharp(\mathcal{L}) < \sharp(\mathcal{T}) = \infty$. As a consequence,

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

    Also, by the Lemmas 11.7 and 11.9, 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 $sw(\mathcal{R}) < sw(\mathcal{R}) + sw(\mathcal{L}) < sw(\mathcal{T})$. In addition, obviously $\sharp(\mathcal{R}) \leq \sharp(\mathcal{T})$. So

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

  - $\mathcal{T}_L$ 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 and $\mathcal{L}$ is finitely failed. In the latter case, as we have seen above, $\mathcal{R}$ is successful.

- $\mathcal{L}$ is successful. By Lemma 11.4(1), (8) and the inductive hypothesis, we conclude that there are $\mathbf{S}_{P_r}$-derivations of the sequents

$$[A \leftarrow A\,;\,R] \Rightarrow B_i \qquad for\ i \in [1, n]. \tag{a}$$

An application of MP gives a derivation of

$$[P] \Rightarrow A \tag{I}$$

- $\mathcal{L}$ is finitely failed, and $\mathcal{R}$ is successful. By Lemma 11.6, (8), and the inductive hypothesis, there is a $k \leq n$ such that the following sequents are derivable in $\mathbf{S}_{P_r}$:

$$[A \leftarrow A; R] \Rightarrow \neg B_k \tag{b}$$
$$[A \leftarrow A; R] \Rightarrow^* B_i \qquad for\ i < k \tag{c}$$

Also, by (9) and the inductive hypothesis, there is a derivation of

$$[R] \Rightarrow A \tag{d}$$

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 Lemma 11.5, (7), and the inductive hypothesis, there are derivations of

  $$[A \leftarrow A\,;\,R] \Rightarrow^* B_i \qquad for\ i \in [1, n]. \tag{f}$$

  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 sequents (b) and (c), for some $k \leq n$. Also, by (6) inductive hypothesis, there is a derivation of the sequent

$$[R] \Rightarrow^* A \tag{g}$$

Thus by an application of PFX_3, we can infer derivablity 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 can infer derivability of $[R] \Rightarrow \neg A$. By (7) and the inductive hypothesis we can infer existence of a $k \leq n$ for which (b) and (c) are 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 claim. □

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 for normal programs are LDNF-trees rather than LD-trees, which complicates the correctness proof for that more general case. We refer the reader to our future work [13], where this extension will be treated in full.

12 Related work

12.1 Linear Logic

Cerrito [8] uses the framework of Linear Logic for a characterisation of Prolog computation on poropositional 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:

$$\begin{aligned} p &\leftarrow \\ p &\leftarrow p \\ r &\leftarrow p, q \end{aligned}$$

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 10.1:

Theorem 12.1 *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 theorey $\mathbf{LT}_P$. 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 neglectable. This can be seen by the following arguments:

1. Linear negation $^{\perp}$ is the only (linear) conective 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. [10].) In fact, it is only contraction with the literals that correspond to the heads of clauses that is harmful.

12.2 Logic as programming, van Benthem

Our approach, using Gentzen style sequent calculi for the characterisation of logic programming styles, originates in [4]. There, a calculus for the general depth first serach procedure was proposed. Intuitive arguments for the soundness and completeness were given. The calculus discussed there included the rules rightward extension and rightward contraction. Also, incorporation of negation on goals was proposed by the use of a second sequent implication $\Rightarrow^-$.

12.3 Andrews

An operational semantics for Prolog, including negation as failure and floundering on non-ground negative goals, is given in [3]. The operational semantics is based on the Clark completion of programs, and has the form of a derivational calculus. The semantics is sound and complete with respect to a 'truth-functional' semantics in the form of a valuation function. This valuation function, defined for the class of goal-formulas, is based on the fold/unfold congruence relation induced by normal programs. Prolog's top-down clause processing and leftmost selection rule are reflected in the use of non-commutative disjunction and conjunction.

12.4 Mints type calculi

Mints calculi [17], [18] are derivational calculi characterising Prolog computation, such that a goal $A = A_1, \ldots, A_n$ succeeds (fails finitely) from the program P via Prolog computation if and only if A $((A)$ is derivable in the calculus for P. The calculi consist of a fixed set of derivation rules and axiom, and a set of axioms describing the program. The calculi deal with predicate programs, and compute the leftmost computed answer substitution. The original system ([17]) incorporated the Prolog 'cut'. Shepherdson [18] extends this set-up to a calculus for normal programs, allowing negation in the bodies of clauses and in goals. Furthermore, [18] gives modified calculi characterizing SLD- and SLDNF-resolution. Essentially, a derivation in a Mints calculus is an attempt to build the relevant part of the (relevant) search-tree—for example, a successful derivation in the calculus for Prolog is isomorhpic to the searchpath the LDNF-tree (cf. Section 11).

12.5 Declarative semantics for Prolog

Stärk, in [19], axiomatises Prolog's leftmost selection rule, by using a new completion of a logic program. The l-completion is formulated as a fist-order theory, in a language extended with three predicates expressing success, finite failure, and left-termination. Success, finite failure and left-termination of a query under Prolog's leftmost selection rule corresponds to provability of the corresponding formula from the l-completion. For correctly moded programs and queries, the l-completion also correctly computes answer substitutions obtained under Prolog computation.

13 Further issues

- As noted before, the calculus $\mathbf{S}_{Pr}$ for standard Prolog computation can be generalised for normal programs. We refer to our future work [13].
- An obvious extension is provided by incorporating the formalisation of various Prolog built-ins. We give some examples:
  - 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 automatically succeeding atom, that is, $\texttt{fail} := \neg\top$.
  - In the context of $\mathbf{S}^*$, the Prolog built-in **;** is implemented by directed disjunction $/$. The built-in **;** is defined as follows: A clause $A \leftarrow R, B\mathbf{;}C, Q$ in a Prolog program P has the same effect as replacing the original clause with $A \leftarrow R, B, Q$, followed by $A \leftarrow R, C, Q$. An easy counterexample shows that this does not generalise to $\mathbf{S}_{Pr}$. That is, in an extension of $\mathbf{S}_{Pr}$ with the introduction rules for directed choice, this connective does not implement the built-in **;**.
  - The built-in predicates **asserta** and **assertz**, effecting their argument to be prefixed, respectively postfixed to the program, can be incorporated by means of introduction rules.
  - Of a different nature is the Prolog cut !, which affects the backtrack procedure in a sophisticated way. It can be incorporated in the calculus for Prolog in a natural way, cf. [13].
- An interesting question is whether the Gentzen format used above is suitable for characterisations of computation procedures on predicate programs. Clearly, the use of reflexive clauses $A \leftarrow A$ as a prefix to programs

in the assumptions of the modus ponens and prefixing rules to prevent divergence on the goal A is specific for the propositional case. An extension of the Gentzen format, suitable for the predicate case, is currently being investigated, and will be reported upon in [5].

- 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 [15] (also cf. [11]) can be conducted for the calculus **S** (cf. [12]). 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 dervation rules atomic monotonicity, Full Context Cut, and Partial Cut for **S** reflect in (untraditional) properties of $C^*_{\mathbf{S}}$. Unlike most nonmonotonic consequence operators, $C^*_{\mathbf{S}}$ is not supranormal.

Acknowledgements The enthousiasm and support of Johan van Benthem have been crucial during my work on this paper. Also, discussions with Grisha Mints have been very helpful. I want to thank the participants in the Compulog project and the Amsterdam Logic Programming Colloquium, and especially Krzysztof Apt, for their comments, and Atocha Aliseda and Makoto Kanazawa for reading draft versions of this paper.

References

[1] K.R. Apt, Logic Programming, in: J. van Leeuwen, Ed., Handbook of Theoretical Computer Science (Vol B), Elsevier, 1990, pp.495–574.

[2] K.R. Apt and D. Pedreschi, Reasoning about termination of pure Prolog programs, Information and Computation **106**(1), 1993, pp. 109–157.

[3] J.H. Andrews, A logical semantics for depth-first Prolog with ground negation, Proceedings ICLP, Vancouver, Canada, 1993.

[4] J. van Benthem, Logic as programming, Fundamenta Informaticae **17**(4), 1992, pp. 285–318.

[5] J. van Benthem and M. Kalsbeek, Gentzen systems for predicate Prolog, in preparation.

[6] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini, Meta for modularising Logic Programs, in: A. Pettorossi, Ed., Meta-programming in logic, Lecture notes in Computer Science 649, Springer, 1992, pp. 105–119.

[7] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini, Modular Logic Programming, ACM Transactions on programming languages and systems, 1994, to appear. Also as Technical report, University of Pisa, 1993.

[8] S. Cerrito, A linear axiomatisation of Negation as Failure, Journal of Logic Programming **12**, 1992, pp.1–24

[9] K. Došen, Modal logic as metalogic, Journal of Logic, Language and Information **1**(3), 1992, pp. 173–202.

[10] J.-Y. Girard, Y. Lafont, and P. Taylor, Proofs and Types, Cambridge University Press, Cambridge, 1990, pp.112–113.

[11] D. Israel, The role(s) of logic in artificial intelligence, in: D. Gabbay, C. Hogger, and J. Robinson, Eds., Handbook of Logic in Artificial Intellingence and Logic Programming (Vol. I), Clarendon Press, 1992, pp. 1–27.

[12] M. Kalsbeek, Computations and operators, manuscript.

[13] M. Kalsbeek, Gentzen systems for Prolog, in preparation.

[14] J.W. Lloyd, *Foundations of Logic Programming*, 1987, Springer.

[15] D. Makinson, General patterns in nonmonotonic reasoning, in: D. Gabbay, C. Hogger, and J. Robinson, Eds., Handbook of Logic in Artificial Intellingence and Logic Programming (Vol. II), Clarendon Press, 1994+.

[16] D. Miller, A logical analysis of modules in Logic Programming, Journal of Logic Programming **6**, 1989, pp. 79–108.

[17] G.E. Mints, Several formal systems of the Logic Programming, Computers and Artificial Intelligence 1990(9), pp. 19–41.

[18] J.C. Shepherdson, Mints type deductive calculi for Logic Programming, Annals of Pure and Applied Logic **56** 1992, pp. 7–17.

[19] R.F. Stärk, The declarative semantics of the Prolog selection rule, in: 9th Annual Symposium on Logic in Computer Science, Paris, 1994, pp. 252–261.

[20] A. Urquhart, Many-Valued Logic, in: D. Gabbay and F. Guenthner, Eds., Handbook of Philosophical Logic (Vol III), Reidel, Dordrecht, 1986, pp. 71–116.