THE UNIVERSITY OF CALGARY

STARLOG: FROM SEMANTICS TO EXECUTION

BY

VINIT NAGARAJA KAUSHIK

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 1991

ISBN 0-315-71189-2

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled *"Starlog: From Semantics to Execution"* submitted by Vinit Nagaraja Kaushik in partial fulfillment of the requirements for the degree of Master of Science.

John G. Cleary

Supervisor, Dr. John G. Cleary
Department of Computer Science

[illegible]

Dr. Graham Birtwistle
Department of Computer Science

[illegible]

Dr. Brian Gaines
Department of Computer Science

[illegible]

Dr. Ali Kazmi
Department of Philosophy

Date June 6, 1991

Abstract

Logic-programming languages allow logical formulae to be executed as programs. These languages are lacking when used to model change. Towards a declarative solution to this problem, Cleary designed a logic-programming language *Starlog*, which uses negation and arithmetic constraints over an explicit, real-valued time.

In this thesis, we introduce Starlog's bottom-up execution, give a procedural semantics to the execution method, and prove its correctness for definite, logic programs. (Definite programs do not use negation.) To demonstrate Starlog's feasibility, we have implemented an interpreter that executes bottom up, provides a notion of real-valued time, supports logical arithmetic, and deduces negations efficiently. Here, we describe the rewrites used for implementing negation and our extension of Cleary's logical, interval arithmetic to the domain of negations. Finally, we compare Starlog with related approaches and suggest directions for shaping it as a practical, logic-programming language.

Acknowledgements

This research has been supported financially by Graduate Teaching and Research Assistantships from the University of Calgary and by my supervisor, Dr. John G. Cleary.

I write this part after having raised and refined the chapters in this thesis. It is a release from the technical, and I feel like thanking life and all humanity for the experience. In fear of indulgence, I shall acknowledge only the major contributions.

I am grateful to John Cleary for opening this window to a fascinating world of logic programming. I thank him for his patience, especially when commenting on draft after draft of this thesis, and for his anecdotes. I still marvel at his clear thought and speech. He would anchor me when I would drift, when Chinooky winters thawed into golden summers.

Thanks are due to Ian Witten for sharpening my writing style and to Alan Dewar and Susan Rempel for helping with a Prolog interpreter.

My thanks go to friends Anja Haman, Rosanna Heise, Zoran Kacic-Alesic, and Vishwa Ranjan for fielding games and extra-curricular activities.

I shall always remember the prompt and cheerful service of the Department's staff. Unseen and unsung, the system administrators at the Department have taken care of computing resources; I appreciate their effort.

I dedicate this thesis to my parents, who have nurtured my scholarly interests.

Contents

List of Figures

Chapter 1

Introduction

Language shapes the way we think,
and determines what we can think about.
— *B. L. Whorf*

Logic-programming languages permit declarative or logical formulae to describe computation and be executed as programs. A problem troubling these languages is that of expressing change, both mutation and persistence. This problem arises in areas as varied as event-driven simulation, databases, operating systems, and hardware verification. *Starlog* [CK91], a new, logic-programming language, is designed to address this problem declaratively. It advocates the use of negation and arithmetic constraints over a real-valued time to express change. Starlog resembles the well-known, logic-programming language Prolog [CM81] in syntax, but executes differently.

In Chapter 2, we provide an overview of the Starlog language. We demonstrate Starlog execution in Chapter 3 as a prelude to the following Chapter 4, which formally defines Starlog execution and proves its semantic correctness. Chapter 4 is the core of this thesis, but might jolt the reader with its sudden recourse to very-formal definitions and proofs. Care has been taken to ensure that it can be safely skipped on the first reading. Chapter 5 discusses Starlog's active negation. Chapter 6 compares Starlog with related approaches. Finally, Chapter 7 takes a look at how we implemented the Starlog interpreter, suggests future work, and presents our conclusions. The rest of this chapter discusses the problem of mutation and persistence to introduce Starlog's approach.

1.1 Mutation and Persistence

Mutation of an object is destructive or in-place assignment to it. For example, by assigning to a variable in a (conventional-language) program, the variable is mutated—its old value is over-written by another one. (We use the word "object" in a general sense, to encompass structures that might be as simple as a variable or as complex as an entire database.) *Persistence* of an object is its value's existence over a period of time, e.g., a variable's value persists from the time an assignment caused it until the next assignment. Mutation and persistence are general phenomena. Therefore, it is important for programming languages to support them.

1.1.1 Approaches to Assignment

Imperative-programming languages offer operators to carry out assignments on variables in a program. Functional-programming languages, such as Lisp [WH81], realize that assignment transgresses their formal basis in the λ-calculus, but for efficiency—both space reuse and fast execution—they offer operators such as `setq` and `set!`. Similarly, many logic-programming languages have sacrificed their declarative semantics and offer operators such as `assert` and `retract` to simulate assignment and to improve efficiency by adding lemmata that store intermediate results for later reuse [Kow85].

Preserving state information is complicated in pure functional and classical-logic languages because their formal calculi—involving λ-expressions and predicates—do not support variables that can be accessed and mutated globally, i.e., by different parts of a program. Destructive assignment requires locations to be associated with variables and a notion of ordered execution; neither requirement is directly supported by classical logic. Therein lies the problem in supporting mutation and persistence. In [Kow85], after discussing the problem of destructive assignment, Kowalski concludes that it is "possibly the single most important problem of logic programming."

Starlog offers a declarative way of simulating destructive assignment. In Starlog, constraints on explicit timestamps are used to enforce the notion of sequencing. We now motivate Starlog's approach by simulating assignment to variables.

1.1.2 Assignment: The Problem

Since arithmetic intervals allow a compact notation, we will use them for illustrating ranges of real values. For example, instead of the conjunction of constraints `(3 > T, T >= 1)` over `T`, we bind `T` to the more compact `[1,3)`. Fig 1.1 depicts the life—in terms of a real-valued time—of a variable 'a'. At time `1.0`, the

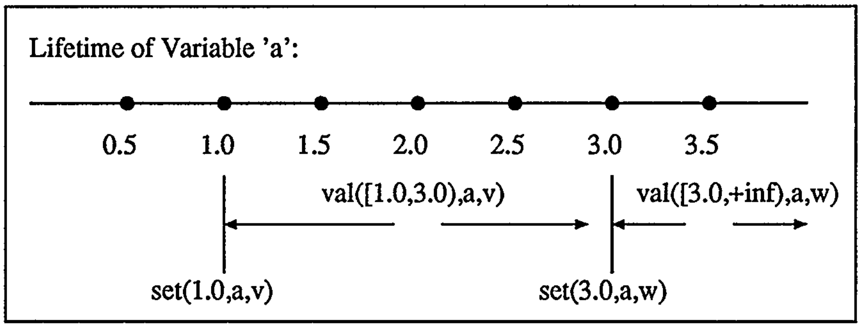


Figure 1.1: A Variable's Lifetime.

variable is assigned or set to a value `v`, and at time `3.0`, it is assigned a value `w`. (We are considering simple, ground values only to simplify this discussion.) From the "definition" of assignment, the variable should have a value `v` at each point in time from `1.0` until—but not including—`3.0`, i.e., the value `v` should persist over the half-open interval `[1.0,3.0)` of real-valued time. (There are many definitions that can be used for assignment. Let us use this one for discussion.) Similarly, the value `w` should persist through each point in time after—and including—`3.0`, i.e., over the time interval `[3.0,+inf)`. (`+inf` and `-inf` denote the positive and negative infinities, $+\infty$ and $-\infty$, limiting the scale of real values.)

1.1.3 Assignment in Starlog

Fig 1.2 shows a program that simulates the assignments in Fig 1.1 and specifies the variable's lifetime. (We generally follow the convention that capital letters be used to

```
% Program Assignment.
% A variable 'Var' has a value 'Val' from---and inclusive of---the time 'Ts',
% when it was set, until---but not including---a time 'Tn', when it is set again.

val(T,Var,Val) <- T >= 0, T >= Ts, set(Ts,Var,Val),
                  not-exists Tn,New: (T >= Tn, Tn > Ts, set(Tn,Var,New)).
set(1.0,a,v).     % at time 1.0, set variable 'a' to a value 'v'.
set(3.0,a,w).     % at time 3.0, set variable 'a' to a value 'w'.
```

Figure 1.2: Program `Assignment`.

denote variables and small letters to denote constants in a program. The symbols '$\leftarrow$' and '`<-`' are used interchangably to mean "implies.") In this program, the rule for `val(T,Var,Val)` states that a variable named `Var` has a value `Val` at a time `T` if there is a time `Ts` in T's present or past—this is the constraint `T >= Ts`—when `Var` was set to a value `Val`. Now, the value `Val` should be given by the most-recent assignment with respect to `T`. So, through the `not-exists` negation in the body, the rule additionally states that there should not be any time `Tn` between `T` and `Ts`—this is through the constraints `T >= Tn` and `Tn > Ts`—when `Var` is set to some value `New`. The `not-exists` construct, which denotes "there does not exist," allows the fresh variables `Tn` and `New` to be introduced locally within the negation.

`val([1.0,3.0),a,v)` and `val([3.0,+inf),a,w)` are assertions of the program in Fig 1.2, when viewed declaratively as a piece of logic. Henceforth called *tuples*, assertions are logical consequences of the program. As we will see later, in Chapter 3, Starlog can execute the program to produce these tuples, as well as the tuples `set(1.0,a,v)` and `set(3.0,a,w)`.

In general, we would like tuples that faithfully extract as much as is specified by the bodies of rules. If an arithmetic variable appears in the head of a rule, we would like its range of values narrowed or squeezed as much as is permitted by the constraints appearing in the rule's body. We would also prefer that solutions be simplified. If a variable `T` is constrained to lie in the interval `[1.0,3.0)`, then instead of solutions that instantiate `T` to each real number in turn in that range, which is infinite, the range itself should be presented. Ranges should also be used, wherever possible, for efficiency. As can be seen from the predicate definition for `val` in Fig 1.2, the positive atom `T >= Ts` in the body can only serve to compute the lower bound of `T`'s interval; the upper bound has to be deduced from the atom `T >= Tn`, which is within a negation. This means that both negation and arithmetic have to be active and logical in Starlog. They must be able to generate bindings and constrain search to make Starlog's execution effective and efficient.

In essence, Starlog advocates the use, over explicit time, of arithmetic constraints and negation to express mutation and persistence. Having briefly introduced Starlog's approach, we take a look at the language itself in the following Chapter 2. For more interesting programs, the reader may refer to Appendix B, [CK91], and [Cle90], which develops a Starlog program that uses sectors to carry out a mixed-mode—continuous-time and discrete-event—simulation of colliding objects.

Chapter 2

Starlog: The Language

Starlog is a logic-programming language that supports arithmetic constraints and a notion of time. Its negation works like a logical constraint—being capable of both receiving and generating bindings (or values) for variables. The rules in a Starlog program can be non-clausal since existentially-quantified variables can be introduced through the `not-exists` predicate, but they must belong to a subclass of first-order formulae that are expressed in prenex conjunctive normal form (PCNF) [Llo87, page 18]. Henceforth in this thesis, we use "rule" to mean an acceptable formula in a Starlog program, "rule-instance" to mean a Starlog formula deduced from a program, and "tuple" to mean the head of a unit rule-instance or assertion deduced from a program.

In this chapter, we discuss Starlog's primitives, syntax, pragmatics, notion of time, and logical arithmetic.

2.1 Primitives, Syntax, and Pragmatics

2.1.1 Primitives

The following are Starlog's declarative, built-in predicates:

- `>`, `>=`, and `=:=` are the "greater than," "greater than or equal to," and "equal to" arithmetic constraints.

- `<`, `<=`, and `!=` are the "less than," "less than or equal to," and "not equal to" arithmetic constraints.
- `+(X,Y,Z)` and `*(X,Y,Z)` are the relational, arithmetic constraints that the sum and product, respectively, of `X` and `Y` is `Z`.
- `real(X)` and `notreal(X)` are the constraints that `X` is or is not real valued.
- `int(X)` is the constraint that `X` is an integer.
- `=` and `=/=` are the general equality and inequality constraints for arbitrary terms.

As yet, there are no primitives for input and output from programs or for user interaction, but declaratively incorporating these and other real-world requirements into Starlog is being considered.

2.1.2 Syntax

Starlog programs follow the Edinburgh-style syntax for Prolog [CM81]. The bodies of rule-instances can be of the following forms:

- a user-defined predicate of the form `p(T,U1,...,Un)` where `T` is a timestamp, and `U1,...,Un` are arbitrary terms. We use the word "timestamp" to mean an arithmetic variable (or value) that can take on only time values.
- a built-in predicate, e.g., `U1 >= U2`.
- a negation of a valid body `L`, e.g., `not(L)`.

- a quantifying negation of the form `not-exists X1,...,Xm: (L)`.

  Here, `X1,...,Xm` are variables quantified existentially *within* the negation, and `L` is a valid body.

- a conjunction of valid bodies, e.g., `(L1, L2, ...)`.

- a disjunction of valid bodies, e.g., `(L1; L2; ...)`.

Appendix A formally specifies Starlog's syntax in Backus-Naur Form (BNF).

2.1.3 Pragmatics

If `p(Th,U1,...,Un)` is the head of a rule-instance, the following constraints should be contained in or obeyed by the body:

- `Th >= 0`. This ensures that the first term of each user-defined predicate is a timestamp.

- `Th >= Tb` for each user-defined predicate `q(Tb,V1,...,Vm)` in any literal, even those within negations, in the body of a rule-instance. This ensures that the program is causal, i.e., the truth of the head depends only on the truth of predicates that are contemporary or earlier in time.

For logic programming, in general, unrestricted use of negation causes serious, semantic problems. Therefore, Starlog restricts its domain to a class of temporally-stratified programs. A sufficient condition for a program to be *temporally stratified* is that each recursive, predicate-call loop involving negation should be accompanied by a time advance, i.e., there should be no zero-delay loops involving negation [CK91].

2.1.4 Programs

The simplest form of Starlog programs is a unit clause, which states that a particular tuple is true. For example, the following clause says that `p(T,a,b)` is true at the instant 1.5:

```
p(1.5,a,b).
```

As will be explained ahead in Chapter 3, Starlog uses a forward-reasoning procedure for execution. Therefore, each program must contain one or more such unit clauses so that tuples, which are the useful products of execution, may be generated.

To assert that a tuple is true over an interval of time, constraints need to be placed on the timestamp. The following clause generates the tuple p((1.5,2.3],c,d):

```
p(T,c,d) <- T > 1.5, 2.3 >= T.
```

To create more-interesting programs, it is necessary to allow more-general conditions in the body of a rule. For example, to say that p will be true 1.5 time units after q, the following clause can be used:

```
p(Th) <- q(Tb), Tb >= 0, +(Tb,1,Th).
```

In this clause, the body (tail) contains one call to a user-defined predicate `q(Tb)` and two calls to Starlog's arithmetic primitives `Tb >= 0` and `+(Tb,1,Th)`. The head (consequent) of the clause is `p(Th)`.

2.2 Logical Arithmetic

Arithmetic in Prolog-like, logic-programming languages is dependent on the order of execution, and errors that are due to limited, floating-point precision are propagated [Cle87]. Also, arithmetic in Prolog is based on functions, and so, it loses the generality for expressing constraints, which are multi-directional relations.

Since Starlog's time is real valued and since Starlog advocates the logical expression of mutation and persistence via constraints on time, it is important that its arithmetic be simple to express, correct, and constructive, i.e., capable of forcing solutions. We chose the logical, interval arithmetic of [Cle87] to implement arithmetic in Starlog. *Internally*, this method represents arithmetic variables as intervals—closed or open—of real numbers and executes relational, arithmetic operations on these intervals. The arithmetic is unharmed by the order of parameter instantiation. Cleary [Cle87] illustrates the expressive power and declarativeness of this method by producing invertible programs that compute factorials and solve general polynomials. We have extended the logical, interval arithmetic of [Cle87] to the domain of negations; this will be clarified in Chapter 5.

2.3 Explicit Time

In Starlog programs, each user-defined predicate is explicitly timestamped. Time in Starlog has the following topology:

- continuous. Time has infinitely-many, real values. (Starlog's implementations may impose bounds on precision and allow only a finite number of values.)

- totally, linearly ordered via the arithmetic operator >=. This is as opposed to the alternatives of branching and circular time [Gal87].

- bounded by a value `0.0`, which is the smallest value time can take, on one side of the ordering. Although this restriction might require some applications to be temporally translated—a minor effort—in order to be programmed in Starlog, it makes Starlog's arithmetic more efficient and constructive. There is no finite, upper bound on time.

The Starlog interpreter uses timestamps to force solutions out of negations—this is explained in Chapter 5—and can use it to schedule execution, and thereby, detect and eliminate tautologies—this is suggested in §7.4.3.

Having taken a look at Starlog as a language, we now move on to Chapter 3, where Starlog's execution will be informally discussed.

Chapter 3

Starlog Execution: An Informal Treatment

In this chapter, we provide a broad overview of Starlog execution. First, we discuss some of the important terms used in this thesis, and then, we illustrate the tree-based execution of simple, Starlog programs. Execution will be formally treated in Chapter 4. In this thesis, the word "interpretation" is used to denote an assignment of truth values to predicates [Llo87, page 12], "execution" is used to denote the execution of programs by an interpreter, and "atom" is used for an atomic formula [Llo87, page 6].

3.1 Background

3.1.1 Model Theory Vs. Proof Theory

Both model theory and proof theory are means of formalizing the "meaning" or semantics of a given (logic) program. A *logic program* is a collection of rules, which are clauses in the case of Prolog programs and possibly non-clausal in the case of Starlog programs. According to [Llo87], a *model-theoretic* view of programs sees the output of a program as a model[1] of its rules. A reply to a query should make the query true in the model given by the program. This is in contrast to a *proof-theoretic* view, wherein the program is a theory based on the first-order, predicate calculus and its rules are axioms or invariants of the theory. In this view, answering a query,

[1]An interpretation wherein the program's rules hold true.

by means of outputs or bindings, constitutes an inference drawn from the program.

Given a program, different model-theoretic views can be adopted; they might be "natural" or "intended." For definite programs, which use only Horn clauses, both Prolog and Starlog subscribe to denotational semantics via a least-fixpoint characterization of least, Herbrand models. Some model-theoretic views will be inadequate. For example, it is not generally possible to show that a conjunction of non-clausal formulae is unsatisfiable, i.e., has no model, when restricting attention to Herbrand interpretations [Llo87, pages 17, 39]. Therefore, more general interpretations need to be considered in such cases. Similarly, different proof-theoretic views are possible for a given program, and if their inference rules are applied in a semantically-incorrect manner, they can produce different results.

These views assume that answering a query or goal is the objective. Starlog takes a different stance. As introduced ahead, its rules of inference are similar to those used in [Llo87, page 38] for computing a program P's minimal model via the least fixpoint of a mapping T_P. Given a program P, Starlog's objective is to compute a set of tuples that when grounded equal P's minimal model. Therefore, Starlog execution mirrors its model-theoretic view—it has no notion of queries, and the "model" itself is the objective.

X-computation, which is formally treated in Chapter 4, is used to execute Starlog programs. X-computation and SLD-resolution [Llo87, pages 40–41] differ greatly in their inference rules, and yet, they produce equivalent results when a fair search is used. Standard Prolog is an execution strategy that uses SLD-resolution and selects the leftmost atom in a goal. It selects clauses to resolve against in their order of textual appearance in the program. Due to this unfair, depth-first search rule, standard Prolog is incomplete—though sound—even for definite programs [Llo87, pages 59–60]. Although the Starlog interpreter might not terminate, we will prove, in Chapter 4, that for definite programs, it is both sound and complete.

3.1.2 Deduction: Bottom Up Vs. Top Down

Consider a query-based, logic-programming system, which aims to provide answers or replies to queries when given the rules of a logic program. There are many ways of deducing answers, but based on the direction of inference, deductive inference can be classified into two extremes:

- *top-down* deduction is query-directed, i.e., the query is used in each inference step. This form of deduction works its way from the query, the "top" level, towards the facts in the program—looking for evidence that supports or contradicts the goal. It is also referred to as goal-driven reasoning and *backward chaining* [BF81, page 198].

- *bottom-up* deduction uses the query only in the last inference step to compute a reply. It builds assertions using the program's facts, the "bottom" level, and works its way towards the query—trying to draw conclusions that are appropriate to the goal. It is also referred to as data-driven reasoning, event-driven reasoning, and *forward chaining* [BF81, page 198].

Standard Prolog performs top-down deduction via SLD-resolution. Since Starlog does not support queries, its deduction is closer to the bottom-up form.

3.1.3 Least-Fixpoint Computation

In [Llo87], a fixpoint of a function T_P is used to characterize a minimal, Herbrand model of a program P. For a definite program P, the least fixpoint lfp(T_P) is shown to be the least, Herbrand model of P. A method for computing the least fixpoint in a bottom-up fashion is defined. This method is based on the result that for any definite program P, the least fixpoint is unique and $\text{lfp}(T_P) = T_P{\uparrow}\omega$, where T_P is a monotonic and continuous function defined over the lattice of subsets of P's Herbrand base B_P. (For a general, stratified, normal program P, T_P has to be modified and lfp(T_P), as computed by a layered version of this method, is only guaranteed to be one of many, minimal models of P.) The method starts with an empty interpretation and applies T_P repeatedly to yield (ground) atoms as logical consequences until an interpretation is reached that is unaffected by further application of T_P. This final interpretation is lfp(T_P). It is possible for the lfp of a definite program to be computed only after infinite applications of T_P.

It would be impractical to implement T_P as defined since it deals with ground (variable-free) interpretations, which are often infinite. So, we replace ground terms in an "interpretation" by general terms containing variables. There are, as well, the following efficiency issues:

- suppose that there is a clause $p \leftarrow q(X), r(X)$ and that T_P has already deduced tuple $r(a)$ to be true. Suppose also that there is a complicated clause Γ with a head $q(Y)$ and whose many assertions will be generated only after many-more applications of T_P. Then, there is no need to repeatedly unify $r(X)$ with $r(a)$ and to test for the unifiability of the atom $q(X)$ with each of the many assertions generated through Γ. Therefore, the body atom $r(X)$ should be unified with $r(a)$; then, for the body atom $q(X)$, the search for unifiers with heads of assertions would be constrained to just the possible assertion $q(a) \leftarrow$. Therefore, maintaining specialized versions of the original program to indicate possible unifications would help efficiency.

- "flat" or unstructured interpretations do not allow direct access to "relevant" atoms. If an atom A in the body of a clause cannot unify with the head of some clause Γ, then A need not be tested for unifiability with atoms introduced via Γ. Therefore, some form of indexing would help efficiency.

- if an atom A has been inferred as a logical consequence of the program and introduced into an interpretation, then there is no need to infer it repeatedly, on each application of T_P. Redundant inferences should be avoided.

3.2 Starlog Execution

We now introduce Starlog's execution, which resembles application of T_P and which also caters to the aforementioned issues of efficiency.

3.2.1 Program `2-3-5`

`Hamming`'s problem [Llo87, page 189] or the `2-3-5` problem is to construct the sorted sequence of positive integers that have no prime factors other than 2, 3, or 5. Consider a simplified version of this problem that allows the integers to be generated in any order and even more than once. The Starlog program in Fig 3.1 solves this version. Its rule `R5` states that multiplying an element `U` in the "sequence" by an ele-

```
% Program 2-3-5.
% The predicate 'seq(T)' will be true at those times that are in the sequence
% 1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,... Multiplying an element in the sequence
% by a factor of 2, 3, or 5 gives another element in the sequence.

(R1) seq(1.0).
(R2) factor(2.0).
(R3) factor(3.0).
(R4) factor(5.0).
(R5) seq(T) <- seq(U), factor(V), *(U,V,T).
```

Figure 3.1: Program `2-3-5`.

ment `V` gives another element `T`. `U` is guaranteed by the predicate `seq(U)` to lie in the Hamming "sequence." `V` is guaranteed to be 2, 3, or 5 by the predicate `factor(V)`. Rule `R1` sets up an initial fact that `1` is an element of the desired "sequence."

An execution of a Starlog program is called an *X-computation.* Starlog executes the program in Fig 3.1 to deduce tuples, which are logical consequences of the program. It selects each rule in the program in turn and generates a tuple or deduces a new rule-instance from it. We shall depict the progress of Starlog execution by incrementally building a tree, called the *X-tree.* The X-tree's nodes are rule-instances, and its edges or arcs, shown as thick lines, link a rule-instance to rule-instances deduced from it. Additionally in this X-tree, each user-defined, predicate call or atom in the body of a rule-instance is associated with a directed *pointer*, shown as a thin arrow. When executing a rule-instance Γ, suppose an atom A is selected. Then, the use of A's pointer is as follows:

> Suppose A points to a rule-instance Δ. Then, only the leaves of the subtree whose root is Δ need to be used as input rule-instances, henceforth called *input-rules*, when deducing from Γ.

Regarding notation, we further use the symbol '$\sim$' to show bindings of variables:

X~[1,3) means that X is bound to the (internal) arithmetic interval [1,3), and Y~f(Z,a) means that Y is bound to the data structure or term f(Z,a). Fig 3.2 summarizes the legend for the diagrams and programs in the rest of this thesis.

Legend:

X~U	Shows that variable 'X' is bound or instantiated to a term 'U'.
━━━	Links parent to child in an X-tree.
- - - - -	Delimits a stage of an X-computation used to construct an X-tree.
⤳	Links an atom in the body of a rule-instance to a rule-instance that it points to in an X-tree.

Figure 3.2: Legend.

Returning back to Fig 3.1, rule R1 is selected first for execution. It is a unit clause, and so, its head is a tuple, and the clause need not be further executed. Similarly, rules R2, R3, and R4 are selected, and their heads are classified as tuples. Now, the non-tuple rule R5 is selected for execution. Since this is an initial execution, each user-defined predicate call in its body is made to point to the root of the X-tree. Fig 3.3 shows the X-tree at this stage in the X-computation; the thin, dashed line delimits successive stages in the X-computation.

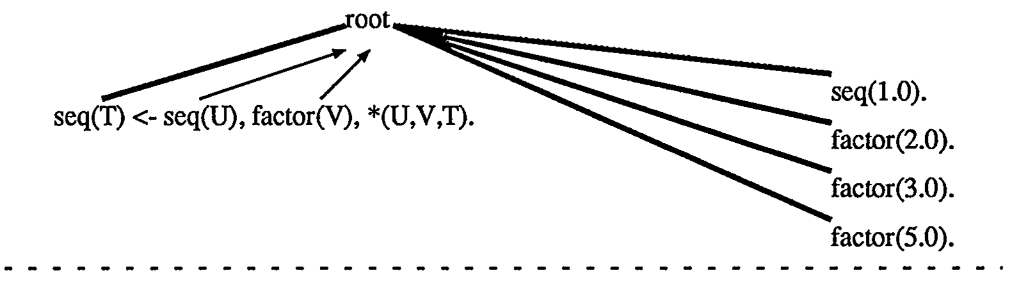


Figure 3.3: Initial X-tree for Program 2-3-5.

Assume that all the atoms in the body of a rule are to be selected, in turn, from left to right. On selection, an atom is to be unified with the head of an input-rule to yield a new rule-instance. Now, consider the execution of rule R5. When the leftmost atom `seq(U)` in R5 is selected, it unifies with the following:

1. R1's head, which is a tuple. This satisfies the selected atom and results in a new rule-instance `seq(T) <- factor(V), *(1,V,T)`. (Ignore renaming of variables, for now.) When this `factor(V)` is selected, it unifies with the following:

   (a) R2's head, which is a tuple. This results in `seq(T) <- *(1,2,T)`, a new rule-instance. Selection of `*(1,2,T)` invokes Starlog's arithmetic built-in, and a tuple `seq(2.0)` is yielded.

   (b) R3's head, which is a tuple. This results in `seq(T) <- *(1,3,T)`, a new rule-instance. Selection of `*(1,3,T)` invokes Starlog's arithmetic built-in, and a tuple `seq(3.0)` is yielded.

   (c) R4's head, which is a tuple. This results in `seq(T) <- *(1,5,T)`, a new rule-instance. Selection of `*(1,5,T)` invokes Starlog's arithmetic built-in, and a tuple `seq(5.0)` is yielded.

2. seq(T), which is the head of the non-tuple rule R5. (Note that R5 is a recursive rule.) No fresh binding can be inferred for U to constrain its values, and the selected atom `seq(U)` remains unsatisfied, resulting in the original rule (modulo renaming). (A model-theoretic, bottom-up form of deduction—instead of resolution—is being used here.) When the next atom `factor(V)` is selected, it unifies with the following:

   (a) R2's head, which is a tuple. This results in `seq(T) <- seq(U), *(U,2,T)`, a new rule-instance. Selection of `*(U,2,T)` invokes Starlog's arithmetic built-in, which is helpless at this stage and returns the same rule-instance.

(b) `R3`'s head, which is a tuple. This results in `seq(T) <- seq(U), *(U,3,T)`, a new rule-instance. Selection of `*(U,3,T)` invokes Starlog's arithmetic built-in, which is helpless at this stage and returns the same rule-instance.

(c) `R4`'s head, which is a tuple. This results in `seq(T) <- seq(U), *(U,5,T)`, a new rule-instance. Selection of `*(U,5,T)` invokes Starlog's arithmetic built-in, which is helpless at this stage and returns the same rule-instance.

Therefore, execution of the non-tuple `R5` resulted in the new tuples `seq(2.0)`, `seq(3.0)`, and `seq(5.0)`. In addition, the following rule-instances await further execution and are placed in a list for scheduling.

```
seq(T) <- seq(U), *(U,2,T).
seq(T) <- seq(U), *(U,3,T).
seq(T) <- seq(U), *(U,5,T).
```

Fig 3.4 shows the X-tree constructed this far and the new links of atoms. Each rule-instance in the X-tree has been standardized apart, i.e., made variable independent. Note that if an atom A is unified with the head of a non-tuple rule Γ, then in the deduced rule-instance, (an instance of) A is made to point to Γ.

The next stage in the X-computation highlights the efficiency gained by using pointers. Refer to Fig 3.5, which shows the result of executing the rule-instance:

```
seq(Ta) <- seq(Ua), *(Ua,2,Ta).
```

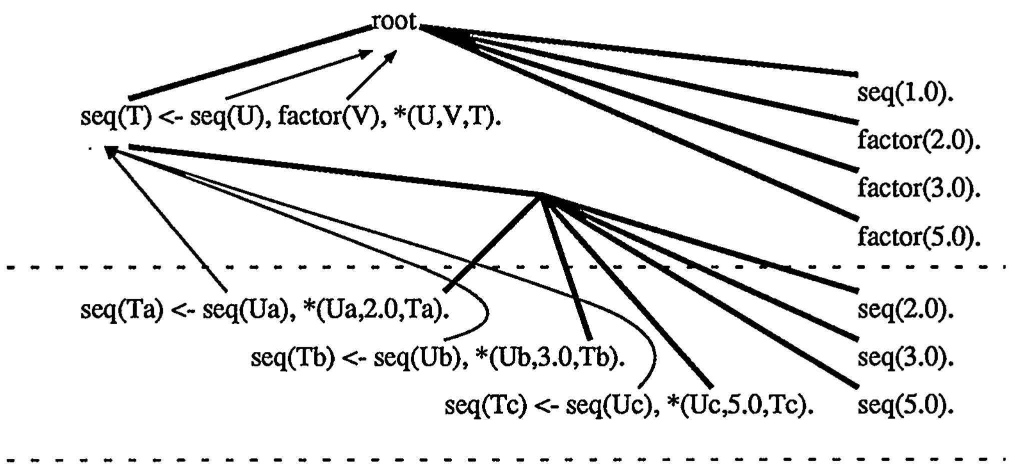

root
seq(T) <- seq(U), factor(V), *(U,V,T).
seq(1.0).
factor(2.0).
factor(3.0).
factor(5.0).
seq(Ta) <- seq(Ua), *(Ua,2.0,Ta).
seq(Tb) <- seq(Ub), *(Ub,3.0,Tb).
seq(Tc) <- seq(Uc), *(Uc,5.0,Tc).
seq(2.0).
seq(3.0).
seq(5.0).

Figure 3.4: Two Stages in an Execution of Program 2-3-5.

Atom seq(Ua) is selected for unification. Each leaf node of the subtree to whose root it points is used, in turn, as an input-rule. This results in six, new rule-instances. In these, when the (further-instantiated) atom *(Ua,2,Ta) is selected, the three tuples seq(4.0), seq(6.0), and seq(10.0) and the following three, non-tuple rule-instances result:

```
seq(Td) <- seq(Ud), *(Ud,2,Td).
seq(Te) <- seq(Ue), *(Ue,2,Te).
seq(Tf) <- seq(Uf), *(Uf,2,Tf).
```

As shown in Fig 3.5, the atoms seq(Ud), seq(Ue), and seq(Uf) point to the non-tuple input-rules used for unification when the atom seq(Ua) was selected. By using pointers, Starlog avoids "reusing" input-rules in a redundant fashion. In this case, specialized, logical consequences of R1 were used, but R1 and R5 were not. In future executions, none of the non-tuple rule-instances generated in this stage will "reuse" the tuples seq(2.0), seq(3.0), and seq(5.0) of the previous stage.

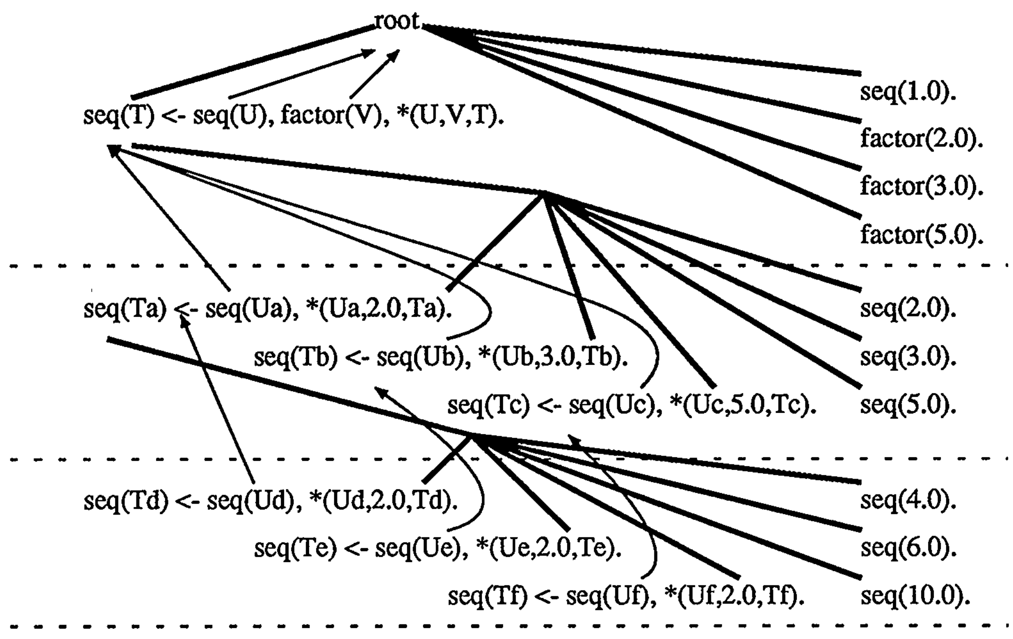

root
seq(T) <- seq(U), factor(V), *(U,V,T).
seq(1.0).
factor(2.0).
factor(3.0).
factor(5.0).
seq(Ta) <- seq(Ua), *(Ua,2.0,Ta).
seq(Tb) <- seq(Ub), *(Ub,3.0,Tb).
seq(Tc) <- seq(Uc), *(Uc,5.0,Tc).
seq(2.0).
seq(3.0).
seq(5.0).
seq(Td) <- seq(Ud), *(Ud,2.0,Td).
seq(Te) <- seq(Ue), *(Ue,2.0,Te).
seq(Tf) <- seq(Uf), *(Uf,2.0,Tf).
seq(4.0).
seq(6.0).
seq(10.0).

Figure 3.5: Three Stages in an Execution of Program 2-3-5.

Starlog execution continues as a sequence of such steps until there are no more non-tuple rule-instances to be executed. The tuples are "returned" as part of Starlog's solution as and when they are computed. In the case of Fig 3.5, Starlog does not terminate in a finite number of steps and the (full) X-tree will be infinite. This is acceptable in view of the infinite nature of the Hamming sequence.

3.2.2 Gleaning Information from Heads

Starlog's bottom-up deduction is a little cleverer than just shown. Suppose the rule-instance `p(T,X) <- q(T,X)` is executed against a non-tuple rule-instance:

```
q(U,f(Y,Z~[1.7,3.2))) <- r(U,Y).
```

(As stated earlier, the latter rule-instance is an input-rule since it is used as input for a deduction step from the former rule-instance.) Then, the following rule-instance is deduced by Starlog:

```
p(Ta,f(Ya,Za~[1.7,3.2))) <- q(Ta,f(Ya,Za~[1.7,3.2)))
```

This means that even partial bindings appearing in the head of a rule-instance R are propagated into the body-atoms that refer or point to R via links in the X-tree; this helps constrain search.

3.2.3 Program `Assignment`

Here, we take up an execution of a program involving negation. Fig 3.7 shows an X-tree for the `Assignment` program, which is reproduced in Fig 3.6. (Fig 3.6 appeared earlier as Fig 1.2.) Since this example is complicated to explain in-line, we urge the

```
% Program Assignment.
% A variable 'Var' has a value 'Val' from---and inclusive of---the time 'Ts',
% when it was set, until---but not including---a time 'Tn', when it is set again.

val(T,Var,Val) <- T >= 0, T >= Ts, set(Ts,Var,Val),
                  not-exists Tn,New: (T >= Tn, Tn > Ts, set(Tn,Var,New)).
set(1.0,a,v).     % at time 1.0, set variable 'a' to a value 'v'.
set(3.0,a,w).     % at time 3.0, set variable 'a' to a value 'w'.
```

Figure 3.6: Program `Assignment`.

reader to refer to Fig 3.8 and Fig 3.9 for tracing through Starlog's deduction of the tuples `val([1.0,3.0),a,v)` and `val([3.0,+inf),a,w)`.

3.2.4 Recapitulation

Given a program P, Starlog's interpreter aims to construct a tree, called the X-tree, whose nodes are rule-instances. When completely built, this X-tree has as leaves tuples that cover P's minimal model, i.e., give P's minimal model when grounded.

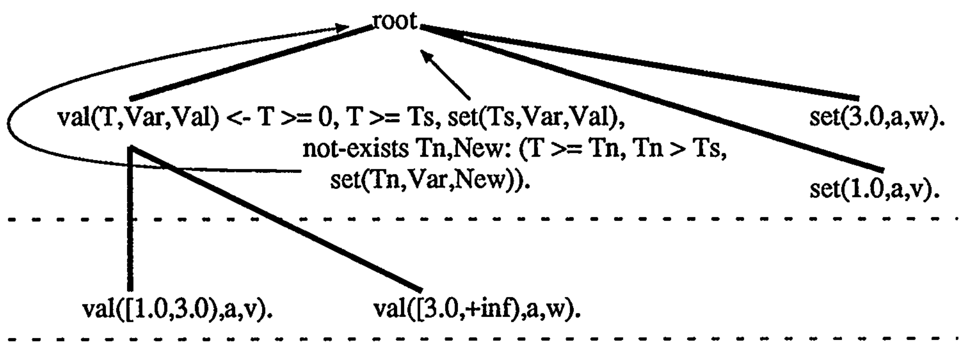

root
val(T,Var,Val) <- T >= 0, T >= Ts, set(Ts,Var,Val),
not-exists Tn,New: (T >= Tn, Tn > Ts,
set(Tn,Var,New)).
set(3.0,a,w).
set(1.0,a,v).
val([1.0,3.0),a,v).
val([3.0,+inf),a,w).

Figure 3.7: A "Compressed" X-tree for Program Assignment.

Each atom in the body of a rule-instance is associated with a pointer to some node in the X-tree, the significance being that exactly those rule-instances that appear at the leaves of the subtree being referred to need to be used as input-rules for deduction with that atom. Initially, the X-tree is a dummy node *root*, whose children are the program's rules; each atom in the body of a program's rule is made to point to the *root*, as in Fig 3.3. The links between parent and child and the pointers between body-atom and head of rule-instance are maintained in the X-tree in order to avoid redundant, deduction sequences.

Starlog first places all the non-tuple rules into a list for scheduling. Execution proceeds by picking a non-tuple rule-instance from the list and applying a form of bottom-up deduction to it. The rule-instances that result are rewritten—according to the rewrite rules introduced later in Chapter 5—and made children of the executed rule-instance in the X-tree, and the non-tuples amongst them are placed in the scheduling list for future execution. Execution terminates when there are no more (non-tuple) rule-instances to be executed. This execution mechanism is very similar to the execution of discrete-event simulations by using event lists [Mis86].

In the previous examples and in the Starlog interpreter that we have implemented, we use *all-atom selection:* when executing a rule-instance, each atom in its body is selected in sequence, from left to right. This is true of even the atoms nested within negations. Although a fair, single-atom selection is sufficient to guarantee correctness of search, all-atom selection is preferable because it improves efficiency by quickening the success or failure of deduction paths in the X-tree. This quickening is because it avoids unnecessary branching, in the X-tree, to create children rule-instances. Atoms that serve to constrain search or fail are picked up immediately—instead of eventually, as would be the case in a fair, single-atom selection. X-trees constructed using all-atom selection are called "compressed" X-trees, to distinguish them from the X-trees got by single-atom selection, which will be formally defined in Chapter 4.

3.2.5 Debugging

Starlog's rule-instances and tuples interact as though they were placed in a shared dataspace or globally-accessible pool. (This is similar to the way computation proceeds in Linda [CG89].) The public nature of communication between rule-instances helps to debug and monitor execution. For example, to fire off an `error` tuple "whenever" the `value` of a modelled variable 'a' is not unique for some period of time `T`, only the following rule needs to be included as part of the `Assignment` program in Fig 3.6:

```
error(T,a,V1,V2) <- val(T,a,V1), val(T,a,V2), V1 =/= V2.
```

This is a means of debugging Starlog programs using Starlog!

3.2.6 X-computations Exploit Parallelism

Each rule-instance in an X-tree can be viewed as a process, and the value of its arguments can be seen as the process's state. X-computations, then, execute multiple processes and in *or*-parallel (concurrent) fashion to search for alternative solutions. As stated in [Sha87, pages 42–43] for the language Concurrent Prolog, a process cannot actively change its state, but can only reduce itself to other processes. Therefore theoretically, Starlog and Concurrent Prolog support only ephemeral processes whose state is not self-modifiable. However, both from an intuitive and an implementation point of view, a process that calls itself recursively with different arguments can be viewed as a *perpetual process* that changes its state. The ability to implement multiple, perpetual processes is one reason for the increased power of Starlog and Concurrent Prolog over (sequential, standard) Prolog, which can implement only one perpetual process without side-effects.

The all-atom selection strategy used by our Starlog interpreter exploits the *and*-parallel nature of programs and can be classified along with the Sync model [LM86], which is categorized in [Con87]:

> AND processes in the Sync model of Li and Martin [LM86] do not use a parallel backtracking algorithm. Instead, they perform an incremental join operation on the values returned by the parallel solution of the literals. Analysis of the body of the rule is used to order the literals, ...

Our Starlog interpreter is a sequential interpreter. Its distributed version would resemble more the model of [LM86] in the execution of conjuncts. Its present execution of literals in a conjunct looks more like pipelining, with the (partial) solutions of one conjunct constraining the solutions of another.

In summary, Starlog uses tuples, which are not grounded as in the application of T_P, for bottom-up deduction. In addition, the tree-based execution scheme uses non-tuple rule-instances to constrain search; also, only "relevant" input-rules are used in each step. Thus, the Starlog interpreter "intelligently" indexes on the entire predicate or atom appearing in the head of a rule-instance. In the following Chapter 4, we will formally define and prove the correctness of Starlog's execution method for definite (Horn-clause) programs.

```
Deduction of Tuple 'val([1.0,3.0),a,v)'.
The following rule is selected for execution:
  val(T,Var,Val) <- T >= 0, T >= Ts, set(Ts,Var,Val),
    not-exists Tn,New: (T >= Tn, Tn > Ts, set(Tn,Var,New)).

Selection of the leftmost constraint in the body satisfies it and gives the following:
  val(T~[0.0,+inf),Var,Val) <- T~[0.0,+inf) >= Ts, set(Ts,Var,Val),
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > Ts, set(Tn,Var,New)).
Selection of the new, leftmost constraint results in the following:
  val(T~[0.0,+inf),Var,Val) <- T~[0.0,+inf) >= Ts~(-inf,+inf), set(Ts~(-inf,+inf),Var,Val),
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > Ts~(-inf,+inf), set(Tn,Var,New)).
Now, the positive 'set' atom is selected and unified with the tuple 'set(1.0,a,v)'.
  val(T~[0.0,+inf),a,v) <- T~[0.0,+inf) >= 1,
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > 1, set(Tn,a,New)).
Since 'Ts' was bound to 1, the "delayed" constraint 'T >= Ts' is woken to give the
following rule-instance:
  val(T~[1.0,+inf),a,v) <- not-exists Tn,New: (T~[1.0,+inf) >= Tn, Tn > 1, set(Tn,a,New)).
---
Now, the negation is selected. The result of executing the following rule-instance
will be used to rewrite the negation:
  h(T~[1.0,+inf)) <- T~[1.0,+inf) >= Tn, Tn > 1, set(Tn,a,New).
Selection of the leftmost constraint, '>=', gives the following:
  h(T~[1.0,+inf)) <- T~[1.0,+inf) >= Tn~(-inf,+inf), Tn~(-inf,+inf) > 1, set(Tn~(-inf,+inf),a,New).
Selection of the next-leftmost constraint, '>', gives
  h(T~[1.0,+inf)) <- T~[1.0,+inf) >= Tn~(1.0,+inf), set(Tn~(1.0,+inf),a,New).
Since 'Tn' was squeezed, the "delayed" constraint 'T >= Tn' is woken to give
  h(T~(1.0,+inf)) <- T~(1.0,+inf) >= Tn~(1.0,+inf), set(Tn~(1.0,+inf),a,New).
The 'set' atom is now selected. It cannot unify with the tuple 'set(1.0,a,v)'.
Unifying it with the tuple 'set(3.0,a,w)' gives
  h(T~(1.0,+inf)) <- T~(1.0,+inf) >= 3.
Now, we have 'Tn~3.0' and 'New~w'.
Binding 'Tn' wakes the "delayed" constraint, '>=', giving the following solution to the body:
  h(T~[3.0,+inf)).
---
Taking this solution of the body back to the negation in 'C' and further rewriting the
binding of the non-local variable 'T' in terms of constraints, we have
  val(T~[1.0,+inf),a,v) <- not-exists Tn~3.0,New~w: (T~[1.0,+inf) >= 3).
Dropping the variables that cannot be further bound from the list of existentially-
quantified variables within the negation, we get
  val(T~[1.0,+inf),a,v) <- not-exists: (T~[1.0,+inf) >= 3).
Since the negation does not existentially quantify any variables, we rewrite as
  val(T~[1.0,+inf),a,v) <- not(T~[1.0,+inf) >= 3).
Since there are no calls to user-defined predicates, we decide to invert the '>='.
  val(T~[1.0,+inf),a,v) <- 3 > T~[1.0,+inf).
The lone constraint in the body is then selected to give the tuple:
  val(T~[1.0,3.0),a,v).
```

Figure 3.8: Deduction of Tuple `val([1.0,3.0),a,v)`.

```
Deduction of Tuple 'val([3.0,+inf),a,w)'.
The following rule is selected for execution:
  val(T,Var,Val) <- T >= 0, T >= Ts, set(Ts,Var,Val),
    not-exists Tn,New: (T >= Tn, Tn > Ts, set(Tn,Var,New)).

Selection of the leftmost constraint in the body satisfies it and gives the following:
  val(T~[0.0,+inf),Var,Val) <- T~[0.0,+inf) >= Ts, set(Ts,Var,Val),
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > Ts, set(Tn,Var,New)).
Selection of the new, leftmost constraint results in the following:
  val(T~[0.0,+inf),Var,Val) <- T~[0.0,+inf) >= Ts~(-inf,+inf), set(Ts~(-inf,+inf),Var,Val),
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > Ts~(-inf,+inf), set(Tn,Var,New)).
Now, the positive 'set' atom is selected and unified with the tuple 'set(3.0,a,w)'.
  val(T~[0.0,+inf),a,w) <- T~[0.0,+inf) >= 3,
    not-exists Tn,New: (T~[0.0,+inf) >= Tn, Tn > 3, set(Tn,a,New)).
Since 'Ts' was bound to 1, the ''delayed'' constraint 'T >= Ts' is woken to give the
following rule-instance:
  val(T~[3.0,+inf),a,w) <- not-exists Tn,New: (T~[3.0,+inf) >= Tn, Tn > 3, set(Tn,a,New)).
---
Now, the negation is selected. The result of executing the following rule-instance
will be used to rewrite the negation:
  h(T~[3.0,+inf)) <- T~[3.0,+inf) >= Tn, Tn > 3, set(Tn,a,New).
Selection of the leftmost constraint, '>=', gives the following:
  h(T~[3.0,+inf)) <- T~[3.0,+inf) >= Tn~(-inf,+inf), Tn~(-inf,+inf) > 3, set(Tn~(-inf,+inf),a,New).
Selection of the next-leftmost constraint, '>', gives
  h(T~[3.0,+inf)) <- T~[3.0,+inf) >= Tn~(3.0,+inf), set(Tn~(3.0,+inf),a,New).
Since 'Tn' was squeezed, the ''delayed'' constraint 'T >= Tn' is woken to give
  h(T~(3.0,+inf)) <- T~(3.0,+inf) >= Tn~(3.0,+inf), set(Tn~(3.0,+inf),a,New).
The 'set' atom is now selected. It cannot unify with either tuple 'set(1.0,a,v)' or 'set(3.0,a,w)'.
Therefore, no tuples can be inferred from the rule-instance for 'h'.
---
Taking this solution of the body back to the negation, we have
  val(T~[3.0,+inf),a,w) <- not-exists Tn,New: (fail).
Since the negation's body is failed, the negation is satisfied and is deleted.
This results in the following tuple:
  val(T~[3.0,+inf),a,w).
```

Figure 3.9: Deduction of Tuple `val([3.0,+inf),a,w)`.

Chapter 4

Formal Semantics for Definite Starlog

Having introduced Starlog execution in an informal manner in Chapter 3, we are now ready to formalize its execution. We restrict ourselves to the class of *definite programs*, which consist of a finite number of single-headed, Horn clauses [Llo87, pages 8–10]; there should be no use of negation in the bodies of clauses. We call this class of programs, along with its execution method in Starlog, "definite Starlog." Restricting to definite Starlog simplifies matters since we do not have to deal with the semantics of negation or arithmetic in Starlog rules. Our results also become generally applicable to Horn-clause programs. Although definite programs lack the expressive power offered by negation, they are an important subset of logic programs since they are computationally adequate, i.e., they are Turing complete [Llo87, Theorem 9.6].

Recall from §3.1.1 that definite Starlog's model-theoretic semantics is given by least, Herbrand models. These models are characterized denotationally by least fixpoints of function T_P [Llo87, pages 37–38]. In this chapter, we formally define definite-Starlog execution and prove that it is semantically both sound and complete with respect to least, Herbrand models. We, therefore, are providing a procedural semantics for definite Starlog.

First, we first introduce our notation. Next, we explicitly assume the use of idempotent unifiers in definite Starlog. Then, we set up apparatus for proving the soundness of definite Starlog. With respect to least, Herbrand models, we define correct tuples, which provide a declarative description of the desired output from a program. The procedural counterpart of a correct tuple is a computed tuple, which is defined using X-computation. We prove that every computed tuple is correct, establishing thereby the soundness of X-computation. Finally, we identify the control elements—of atom- and clause-selection rules—governing X-computation during its search for tuples and prove that every correct tuple is an instance of a computed tuple. This establishes the completeness of X-computation. Therefore, we have the final result that a fair X-computation produces only and all correct tuples.

4.1 Notation and Terminology

Unless redefined, the terminology used here has been borrowed from [Llo87]. LHS and RHS are abbreviations for "left hand side" and "right hand side" of an implication; TPT abbreviates "to prove that." Upper-case Greek letters, except Φ and Ψ, are reserved for clause-instances (defined ahead), lowercase Greek for substitutions, and upper-case Arabic—optionally subscripted with lowercase Arabic letters or numerals—for atoms in clauses; the upper-case Arabic letter P is an exception and always denotes the definite program in question. The upper-case Greek letters, Φ and Ψ, are reserved for naming sets of clause-instances. Lists, which are sequences or ordered collections, are enclosed in box brackets, [and]; sets, which are unordered collections are enclosed in curly brackets, { and }. (Note that [Llo87, page 44] uses $[A]$ differently to mean the set of ground instances of atom A.) Overlined, lowercase Arabic letters or numerals, e.g., $\vec{\imath}$, are reserved for names of (pointers to) clause-instances. We show the names of clause-instances being referred to only when essential to the discussion. The symbol $\Im$—optionally subscripted with lowercase Arabic letters or numerals—is reserved for the names of X-trees (defined ahead), the symbol ∂—optionally subscripted with an upper-case, Greek symbol denoting some clause-instance—is reserved for the names of X-derivations (defined ahead), the symbol $\mho$ is reserved for X-computations (defined ahead), and the symbol $\Re$ is reserved for selection rules.

B_P denotes the Herbrand base [Llo87, page 16], and M_P denotes the least, Herbrand model [Llo87, page 36] of a definite program P. If Γ is a clause, then $\Gamma+$ is its head, and $\Gamma-$ is its body. The abbreviation "mgu" stands for "most general unifier." We use "suitable variant" to mean a variant that is standardized apart, i.e., a variant that is variable independent from the clause-instances in question.

4.2 Idempotent Unifiers: An Assumption

We assume that definite Starlog's unification algorithm incorporates the occur check [Llo87, page 24] and produces only idempotent mgus. This is because in certain proofs ahead, e.g., in Theorem 31, we have to consider idempotent substitutions. Such mgus can be constructed as in [Llo87, page 24]. (For the set $S = \{a, a\}$, both the idempotent unifier $\alpha = \{\}$ and the non-idempotent unifier $\beta = \{X/Y, Y/X\}$ are mgus.) Since mgus are unique modulo renaming [Llo87, page 23], our assumption does not compromise on correctness with respect to unification.

4.3 Soundness of Definite Starlog

Given a program P, Starlog's objective is to compute a (possibly-infinite) set of tuples that when grounded equal P's minimal model. Each tuple is an instance of the head of some clause in P. Tuples can have variables, i.e., they need not be ground. Starlog uses X-computation to compute tuples.

Our aim is to prove definite Starlog's soundness. First, we define correct tuples, which provide a declarative description of the desired output from a program. The procedural counterpart of a correct tuple is a computed tuple, which is defined using X-computation. X-computation enhances the method for computing least fixpoints in [Llo87] by additionally using pointers and non-unit input-clauses when executing bottom up. We prove that every computed tuple in an X-computation is a correct tuple. This establishes the soundness criterion.

Pointers as Names: In Chapter 3, we have introduced pointers from atoms in the bodies of rule-instances to rule-instances in the X-tree. For this chapter, we shall represent these pointers using names. The named objects are the clause-instances of P. Each atom A in the body of a clause-instance refers to some clause-instance via the latter's name, say, $\vec{n}$; this reference is depicted by $A^{\vec{n}}$. $\vec{n}$ is said to be the *pointer* associated with or of A. We use a countably-infinite, absolute name space, wherein names uniquely identify the objects being named. (This is in contrast to a possible relative name space wherein a context may be required, in addition to the name, to uniquely identify an object, e.g., file-names on Unix$^{\text{TM}1}$, which need the context provided by the full path from the file-system's root node in order to uniquely identify

^{1}Unix is the trademark of AT&T Bell Laboratories.

a physical file.) Names may be hierarchical or flat without affecting our results. We take $\mathcal{N} = \{\vec{0}, \vec{1}, \vec{2}, \ldots\}$ as our flat name space here. The named objects are the clause-instances of P. Let $\vec{0}$ be reserved to refer to the *root* of the program's X-tree (in Def 14 ahead).

Now, we launch into formal treatment.

Def 1 *An atom A is a* correct tuple *of P iff $\forall(A)$ is a logical consequence of P.*

Any further instantiation of a correct tuple always gives a correct tuple.

Def 2 *Deleting or dropping zero or more atoms in the body of a clause constitutes* atom dropping.

If $\Gamma = H \leftarrow A_1, \ldots, A_m, \ldots, A_k$ then an example clause got by atom dropping Γ is $H \leftarrow A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k$.

Def 3 *Changing zero or more pointers associated with atoms in the body of a clause constitutes* pointer changing.

If $\Gamma = H \leftarrow A_1^{\vec{a}}, \ldots, A_m^{\vec{c}}, \ldots, A_k^{\vec{e}}$ then an example clause got by pointer changing Γ is $H \leftarrow A_1^{\vec{a}}, \ldots, A_m^{\vec{c'}}, \ldots, A_k^{\vec{e}}$; here, $\vec{c'}$ may or may not be $\vec{c}$. All the pointers referred to are assumed to be names of valid clauses. This is to avoid unnecessarily complicating the discussion.

Def 4 (Clause-instance, Stage) *Let Γ be a definite clause. Then, there exists a minimal (non-empty) set Φ of all* clause-instances *of Γ such that:*

- *Γ is in Φ, and*

- *Φ is closed under atom dropping, pointer changing, and substitution.*

Any subset of Φ is called a stage. *The union of sets of clause-instances of different clauses is also a stage.*

The stage consisting of exactly the clauses in P such that each atom in the body of a clause refers to $\vec{0}$ is given a special name *init*, signifying the initial stage in P's X-computation. A clause-instance of a clause in P is sometimes called a clause-instance of P. We use "clause" to mean a clause in P and "clause-instance" to mean any clause-instance of P.

From now on, we will need two naming functions: $\mathcal{ID}$ and $\mathcal{CL}$. $\mathcal{ID}$ is the name binder, and $\mathcal{CL}$ is the name resolver. Our use of an absolute name space guarantees that these are bijections. We take them as inverse bijections of one another: given a clause-instance, $\mathcal{ID}$ provides a name, and given a name, $\mathcal{CL}$ provides a clause-instance. Formally stated, if S is the set of all clause-instances of P then $\mathcal{ID}: S \mapsto \mathcal{N}$, and $\mathcal{CL}: \mathcal{N} \mapsto S$.

Def 5 *Let* $\Gamma = H \leftarrow A_1, \ldots, A_m^{\vec{c}}, \ldots, A_k$ $(k \geq 0)$, $\Delta = F \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$, *and* $\mathcal{ID}(\Delta) = \vec{h}$. *Then* Γ' *is* derived from Γ using Δ *iff all the following hold:*

- $k > 0$ *and* A_m *is an atom, called the* selected atom, *in* $\Gamma-$.
- *there exists an mgu* α *for* A_m *and* F', *a suitable variant of* F, *i.e.,* $\Rightarrow A_m\alpha = F'\alpha$.
- *if* $n > 0$ *then* Γ' *is a suitable variant of* $(H \leftarrow A_1, \ldots, A_m^{\vec{h}}, \ldots, A_k)\alpha$. *If* $n = 0$, *which means* Δ *is a unit clause, then* Γ' *is a suitable variant of the clause-instance* $(H \leftarrow A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\alpha$.

Δ *is called the* input-clause *for the derivation of* Γ' *from* Γ.

The pointers of unselected atoms cannot change during a derivation step.

Def 6 *Let* Γ *be a clause-instance of* P*. A sequence consisting of only* Γ *is an* X-derivation *from* Γ*. A (possibly infinite) sequence* $[\Gamma_0(=\Gamma), \Gamma_1, \ldots]$ *of clause-instances of* Γ *is an* X-derivation from Γ *iff all the following hold:*

- *each* Γ_{i+1} *is derived from* Γ_i*.*
- *if* $A^{\vec{n}}_m$ *is a selected atom in the derivation of* Γ_{i+1} *from* Γ_i *then the input-clause* Δ_i *in that derivation step is X-derivable from* $\mathcal{CL}(\vec{n})$*.*

We refer to this process as X-execution *of* Γ*.*

Each clause-instance in the X-derivation from Γ is said to be yielded, X-derived, or X-derivable from Γ. Abusing usage slightly, if a tuple B is X-derived from Γ, i.e., the unit clause-instance $B \leftarrow$ is X-derived from Γ, we say that B is yielded by an X-derivation from Γ. An X-derivation involves ≥ 0 derivations, each of which has to additionally satisfy the aforementioned restriction on the origin of the input-clause. Note that each Δ_i, in turn, is required to be X-derivable.

All the clauses in P are assumed to be X-derivable from $\mathcal{CL}(\vec{0})$ and are written with each atom in the body referring to $\vec{0}$. This will be clearer after Def 10 ahead.

Def 7 *The* length *of an X-derivation from a clause-instance* Γ *is the number of clause-instances of* Γ *in it.*

The minimum length of an X-derivation is one. Every clause-instance is X-derivable in unit length from itself.

Corollary 8 (Non-Closure Under Composition) *The composition of unit-length X-derivations does not result in an X-derivation.*

The composition of two unit-length X-derivations would be expected to be of length two. This resultant should have used some input-clause (see Def 6). No input-clause has been used in the two original, unit-length X-derivations, and hence, we have a contradiction. Furthermore, the composition of X-derivations of non-unit length does not always result in an X-derivation because the terminating and initial clause-instances in the two X-derivations have to "match" at the point of composition.

Def 9 *Let* Γ *be a definite clause in* P. *An* X-assertion *from* Γ *is a finite X-derivation from* Γ *whose sequence of clause-instances ends in a unit clause.*

A finite X-derivation may be successful or failed. A *successful* X-derivation is an X-assertion. Let Γ be a definite clause and Γ' be a clause-instance of Γ that is also X-derivable from Γ. If Γ' is not a unit clause and there is no clause-instance of Γ that is X-derivable in *non-unit* length from Γ', then the X-derivation from Γ ending in Γ' is said to be *failed.*

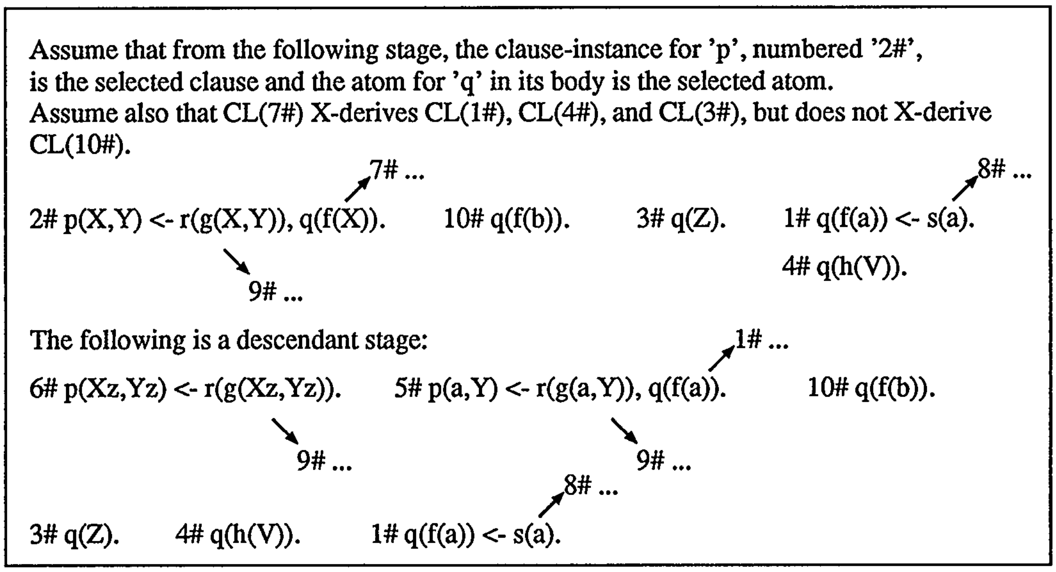
Def 10 *If* $\Phi = \{root\}$ *then init is said to* descend *from* Φ, *and root is said to* bear *each clause-instance in init. If* $\Phi = \{\Gamma_1, \ldots, \Gamma_m, \ldots\}$, *where each* Γ_i *is a clause-instance of* P, *is a stage then* Φ' descends *from* Φ *iff the following hold:*

- Γ_m *is a non-unit clause-instance, called the* selected clause, *in* Φ.
- *let* $A_n^{\vec{c}} \in \Gamma_m-$ *be the selected atom.*

– *if there is no clause-instance that occurs in Φ, is X-derivable from $\mathcal{CL}(\vec{c})$, and whose head is unifiable with A_n then $\Phi' = \{\Gamma_1, \ldots, \Gamma_{m-1}, \Gamma_{m+1}, \ldots\}$, and we say Γ_m* fails.

– *otherwise, let $\Delta_1, \ldots$ be the (potentially-infinite) clause-instances that occur in Φ, that are X-derivable from $\mathcal{CL}(\vec{c})$, and each of whose heads Δ_i+ is unifiable with A_n. Let $\Gamma_{m_1}, \ldots$ be (suitable variants of) the clause-instances derived from Γ_m using $\Delta_1, \ldots$ as input-clauses. We say Γ_m* bears *each of $\Gamma_{m_1}, \ldots$ The descendant Φ' is got by replacing Γ_m in Φ by all the clause-instances in $\{\Gamma_{m_1}, \ldots\}$.*
$\Rightarrow \Phi' = \{\Gamma_1, \ldots, \Gamma_{m-1}, \Gamma_{m_1}, \ldots, \Gamma_{m+1}, \ldots\}$.

Fig 4.1 shows an example, descendant stage. Note that unifiability of a clause-


Assume that from the following stage, the clause-instance for 'p', numbered '2#', is the selected clause and the atom for 'q' in its body is the selected atom.
Assume also that CL(7#) X-derives CL(1#), CL(4#), and CL(3#), but does not X-derive CL(10#).

The following is a descendant stage:

Figure 4.1: Descendance: An Example.

instance's head with $A_n^{\vec{c}}$ is not implied by the former's X-derivability from $\mathcal{CL}(\vec{c})$. If an atom $A^{\vec{n}}$ is selected for derivation when computing the descendant for a given stage, then only the clause-instances that are X-derivable from $\vec{n}$ and that appear in the given stage have to be considered as candidate input-clauses; the search space for input-clauses has thus been pruned from both "above" and "below." Also, the descendants can be computed in a layered or stratified fashion. Each stage may have many possible descendants, which differ in the selection—in some derivation step—of either a clause-instance or an atom within a selected clause-instance. Given a finite stage, each of its descendant stages is guaranteed to be (modulo variants) finite simply because each input-clause, which is used to compute the descendant, has to appear in the given stage.

Def 11 *An* X-computation *of P consists of a (possibly-infinite) sequence:*

$$[\Phi_0(=\{root\}), \Phi_1(=init), \ldots].$$

Each Φ_i $(i > 0)$ is a finite stage; each Φ_{i+1} $(i \geq 0)$ descends from Φ_i. Complete *X-computations are those that are either infinite or that finitely terminate at a stage that has no descendants.*

An X-computation can be viewed as a mapping that operates on stages, analogous to T_P operating on Herbrand interpretations. For a given program P, there could be many X-computations, which differ in at least one stage. Each X-computation carries out or weaves through concurrent threads of X-derivations from clause-instances in its stages. From some clause-instances of P, there could be X-derivations that are not carried out by any X-computation of P. X-computations terminate at the first stage that is either empty or made up of only unit clauses, i.e., when there are no more descendants. Although for certain programs, complete X-computations do not terminate, Starlog attempts to carry an X-computation to completion. The following program is one such that continues forever when executed by Starlog:

```
p(f(X)) <- p(X).
p(a).
```

Def 12 *An atom A is said to be a* computed tuple *of P iff there exists an X-computation $\mho$ of P such that the unit clause-instance $A \leftarrow$ occurs in some stage of $\mho$.*

Equivalently, A should be yielded by some X-assertion ∂ from a clause in P, and ∂ should be carried out by $\mho$. Further instantiation of a computed tuple need not give a computed tuple; computed tuples have to be "exactly" X-assertible. Both the computed and the correct tuples can be non-ground, i.e., they can have variables. Starlog is computationally interested in tuples, which are asserted from the entire program, rather than in answers, which are bindings to variables in a main query or goal.

Def 13 *The* success set *SS_P of P is the set of all $A \in B_P$ such that $A = A'\alpha$, for some computed tuple A' and ground substitution α.*

Note that SS_P may be got by grounding tuples computed by different X-computations of P.

Def 14 *An* X-tree *for P is a tree that is based on a complete X-computation $\mho$ of P and satisfies the following properties:*

- *each node—except the root—in the tree is a definite (possibly-unit) clause-instance of P.*

- *the root node is named* $\vec{0}$.
- *each node in this tree has as children the clause-instances that it bears in* $\mho$. *Nodes that are unit clauses have (bear) no children.*
- *a node that fails signifies the failure of the X-derivation path it appears on.*

In an X-tree, each node is distinct from every other node; the process of X-execution ensures that no variables are shared across nodes. X-trees offer a different view of complete X-computations without placing additional restrictions on them. Each path in the X-tree is an X-derivation from some clause in P. Paths corresponding to successful X-derivations (X-assertions) are called *success paths*, paths corresponding to infinite X-derivations are called *infinite paths*, and paths corresponding to failed X-derivations are called *failure paths.* Note that these X-trees are complete—unlike the (partial) trees constructed, descendant by descendant, in Chapter 3.

Example Consider the definite program `Pqr1` in Fig 4.2. As shown by Fig 4.3

```
% Program 'Pqr1'.
p(X) <- q(X).
q(Y) <- r(Y).
r(1).
```

Figure 4.2: Definite Program `Pqr1`.

and Fig 4.4, both finite and infinite X-trees are possible for `Pqr1`.

Corollary 15 *For each complete X-computation of P, there is at least one X-tree of P based on it. For each X-tree of P, there is at least one complete X-computation of P on which it could be based.*

Tuples, which are yielded by finite paths in an X-tree $\Im$, are eventually computed in each X-computation on which $\Im$ is based. The stages of a complete X-computation

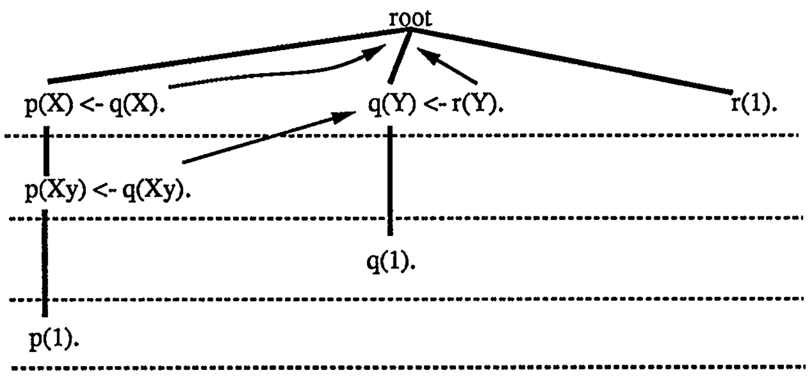


Figure 4.3: A Finite X-tree for `Pqr1`.

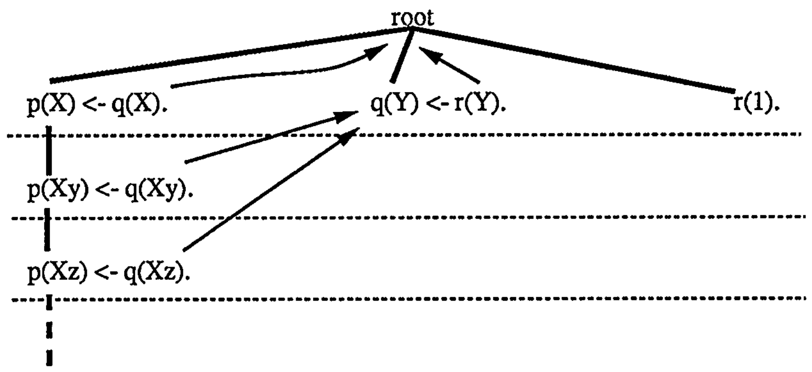


Figure 4.4: An Infinite X-tree for `Pqr1`.

are the sets of leaves, i.e., the yield, during the construction of the X-tree based on it, descendant by descendant. It is easy to see that for each complete X-computation, there is exactly one X-tree of P based on it, but for our proofs, we need only the weaker statement expressed in Corollary 15.

Now, we define a notion of depth in an X-tree. This will be used ahead to prove properties about X-trees.

Def 16 *Consider an X-tree $\Im$ containing a finite X-derivation ∂_Γ from a clause-instance Γ. In $\Im$, the* depth $\mathcal{D}(\partial_\Gamma)$ *of ∂_Γ is defined recursively as follows:*

- *if ∂_Γ is of unit length, then $\mathcal{D}(\partial_\Gamma) = 0$.*
- *otherwise, let ∂'_Γ be that initial subsequence of ∂_Γ that ends in the penultimate clause-instance of ∂_Γ, and let ∂_Δ be the X-derivation in $\Im$ from some clause in P yielding the last input-clause Δ used in ∂_Γ. Then,*

$$\mathcal{D}(\partial_\Gamma) = \mathcal{D}(\partial'_\Gamma) + \mathcal{D}(\partial_\Delta) + 1.$$

One more than the combined depths of the X-derivations yielding Γ and Δ gives the depth of the clause-instance derived from Γ using input-clause Δ. Note that ∂_Γ can have different depths depending on the X-derivations ∂_{Δ_i} selected for each input-clause Δ_i. By fixing the X-tree $\Im$, we are guaranteed that the clause-instances corresponding to its nodes are variable independent and that their X-derivations and those of the input-clauses are fixed and appear in $\Im$. Therefore, for any X-derivation ∂_Γ in an X-tree, $\mathcal{D}(\partial_\Gamma)$ is uniquely defined.

Now, we come to the first of our results. This result will be used to prove definite Starlog's soundness.

Theorem 17 *Consider a stage Φ of an X-computation of P. Each clause-instance in Φ is a logical consequence of P.*

Proof Let Γ_{i+1} be a clause-instance in Φ. Then, there exists an X-derivation ∂_Γ from some clause Γ in P (using Def 11) such that ∂_Γ is carried out by an X-computation of P and such that $\partial_\Gamma = [\Gamma_0(=\Gamma), \Gamma_1, \ldots, \Gamma_{i+1}]$. Therefore, there must exist some X-tree $\Im$ of P in which ∂_Γ occurs as a subpath (using Corollary 15). Let $d = \mathcal{D}(\partial_\Gamma)$ in $\Im$. We induce on d TPT Γ_{i+1} is a logical consequence of P.

Basis: $d = 0$.

$\Rightarrow \partial_\Gamma = [\Gamma_{i+1}] = [\Gamma]$.

$\Rightarrow \Gamma = \Gamma_{i+1}$ is a unit clause in P (using Def 16).

$\Rightarrow$ Basis is true.

Hypothesis: Assume that for $0 < d \leq k-1$ $(k > 1)$, the Theorem is true.

Induction: $d = k$. In $\Im$, let Γ_{i+1} be derived from Γ_i using an input-clause Δ, and let ∂_{Γ_i} be that initial subsequence of ∂_Γ that ends in the clause-instance Γ_i.

$\Rightarrow$ In $\Im$, there exists an X-derivation ∂_Δ from some clause in P yielding Δ (using Def 10).

$\Rightarrow d = \mathcal{D}(\partial_{\Gamma_i}) + \mathcal{D}(\partial_\Delta) + 1$ (using Def 16).

$\Rightarrow \mathcal{D}(\partial_{\Gamma_i}) \leq k-1$ and $\mathcal{D}(\partial_\Delta) \leq k-1$.

$\Rightarrow$ Both Γ_i and Δ are logical consequences of P (using Hypothesis).

Now we shall work through Def 5. Let $\Gamma_i = H \leftarrow A_1, \ldots, A_m, \ldots, A_r$ $(r > 0)$, $\Delta = F \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$, A_m be the selected atom, and α be the mgu for A_m and F', the suitable variant of F. Here, $r > 0$ since Γ_{i+1} is derived from Γ using an input-clause (using Def 5).

- if $n > 0$ then Γ_{i+1} is a suitable variant of $(H \leftarrow A_1, \ldots, A_m, \ldots, A_r)\alpha$, which is certainly a logical consequence of Γ_i, and hence, of P. In this case, the Theorem is true irrespective of α.

- if $n = 0$ then Δ is the unit clause $F \leftarrow$, and Γ_{i+1} is a suitable variant of $(H \leftarrow A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_r)\alpha$. Effectively, Γ_{i+1} is a resolvent.

  $\Rightarrow \Gamma_{i+1}$ can be deduced from Δ and Γ_i, and so, is a logical consequence of P.

  $\square$

Theorem 18 (Soundness of X-computation) *Every computed tuple of P is a correct tuple of P.*

This result establishes the soundness of definite Starlog.

Proof Consider a computed tuple A of P.

$\Rightarrow$ There exists an X-computation $\mho$ of P such that $A \leftarrow$ occurs in some stage of $\mho$ (by Def 12).

$\Rightarrow A \leftarrow$ is a logical consequence of P (using Theorem 17).

$\Rightarrow \forall(A)$ is a logical consequence of P.

$\Rightarrow A$ is a correct tuple of P (using Def 1). $\square$

Corollary 19 ($SS_P \subseteq M_P$) *The success set of a definite program is contained in the program's least, Herbrand model.*

Proof Let P be a definite program, SS_P be its success set, and M_P be its least, Herbrand model. TPT $\forall A \in B_P,\ A \in SS_P \rightarrow A \in M_P$.

$\Rightarrow$ TPT $\forall A \in B_P,\ A \in SS_P \rightarrow A$ is a logical consequence of P (using Theorem 6.2 in [Llo87, page 37]). Let LHS be the proposition $A \in B_P \ \wedge \ A \in SS_P$ and RHS be the proposition that A is a logical consequence of P.

LHS $\Rightarrow$ There exists a computed tuple A' such that $A = A'\alpha$, for some ground substitution α (using Def 12).

$\Rightarrow A'$ is a correct tuple (using Theorem 18).

$\Rightarrow \forall(A')$ is a logical consequence of P (using Def 1). Since A is a ground instance of A', the RHS is true. $\square$

4.4 Completeness of Search

In §4.3, we set up the apparatus for a procedural semantics of definite Starlog and proved its soundness. We now identify the control elements in definite-Starlog execution and prove that its search for tuples is complete with respect to least, Herbrand models. Our aim, therefore, is to prove that every correct tuple is an instance of some computed tuple.

Def 20 *Let Φ be the stage containing all clause-instances of P. An* atom-selection rule *is a function from Φ to a set of atoms such that the value of the function for a clause-instance is an atom, called the* selected atom, *in the body of that clause-instance.*

In theory, there need be no ordering on atoms in the body of a clause-instance and on clause-instances in a stage. In practice, as in our Starlog interpreter, both orderings are imposed to simplify scheduling.

Def 21 *Let Γ be a clause-instance of P and $\Re_A$ be an atom-selection rule. An* X-derivation *from* Γ via $\Re_A$ *is one that uses $\Re_A$ to select atoms.*

An X-assertion from Γ via $\Re_A$ is defined similarly.

Def 22 *An* X-computation *of P* via *an atom-selection rule $\Re_A$ is one that uses $\Re_A$ to select atoms.*

An X-tree via $\Re_A$ is similarly defined.

Def 23 *Let $\Re_A$ be an atom-selection rule. An atom A is an* $\Re_A$-computed tuple *of P iff there exists an X-computation $\mho$ via $\Re_A$ of P such that the unit clause-instance $A \leftarrow$ occurs in some stage of $\mho$.*

A is yielded by an X-assertion that is carried out by an X-computation via $\Re_A$. Note that this is stricter than asking for just the X-assertion yielding A to be via $\Re_A$; this restriction is imposed only to simplify the proofs.

Def 24 *Let* $\Re_A$ *be an atom-selection rule. The* $\Re_A$-success set *of* P *is the set of all* $A \in B_P$ *such that* $A = A'\alpha$*, for some* $\Re_A$*-computed tuple* A' *and (ground) substitution* α*.*

Def 25 *A* search rule *is a strategy for searching or constructing X-trees.*

We explore here an *or*-parallel search of the X-tree of a program P, *or*-parallel in that different paths of this tree are constructed concurrently, i.e., in an interleaved fashion. Concurrency in the search is essential for completeness since the input-clauses for derivations on one path may be X-derived on other paths. Starting with *root*, Starlog constructs the X-tree downwards; computed tuples appear at the leaves of the X-tree as it is being constructed. It may take forever before *all* the computed tuples appear on the X-tree being constructed. This means that a set of all computed tuples of a program may not be finitely computable by Starlog. For example, Starlog takes forever to compute the following program's tuples:

```
    p(f(X)) <- p(X).
    p(a) <-.
```

Unfortunately, Starlog cannot deduce that tuples `p(a)` and `p(f(X))` would, when grounded, equal the program's minimal model. This means that even if there is a finite set of tuples that when grounded equals a program's minimal model, Starlog might not terminate finitely.

Def 26 *Let Φ be the set of all clause-instances of P. A* clause-selection rule $\Re_C$ *is a function from 2^Φ to Φ such that given a stage, $\Re_C$ selects (decides) a single non-unit clause-instance from that stage.*

Def 27 *A* fair clause-selection rule $\Re_C$ *is one that guarantees that each non-unit clause-instance in an input stage is eventually selected in each X-computation using $\Re_C$.*

A fair $\Re_C$ offers a non-zero, though not necessarily equal, probability of selection to each clause-instance in a given stage. In a stage of an X-computation, i.e., leaves of the corresponding X-tree as it is being constructed, a clause-instance upon X-execution is replaced by some different clause-instances, each of which will be selected eventually by a fair $\Re_C$. Fairness of clause-selection ensures that each finite path in the X-tree is fully enumerated within a finite number of X-computation stages; infinite paths may only be partially enumerated. A first-come-first-served (FCFS) clause-selection rule is fair and is used by our Starlog interpreter.

We will prove ahead that fair clause-selection rules are semantically equivalent. (In practice, depending on P, some fair clause-selection rules may perform better than others, e.g., by computing tuples in fewer steps or by reducing branching in the X-tree.) We are interested in fair rules since they guarantee—as proved ahead in Theorem 36—Starlog's completeness and not just soundness. Note that the X-tree for a program is dependent on both the atom-selection and clause-selection rules.

Def 28 *Let $\Re_A$ be an atom-selection rule and $\Re_C$ be a clause-selection rule. An* X-computation $\mho$ *of P* via $\Re_A$ *and $\Re_C$ is an X-computation of P in which $\Re_A$ is used to select atoms and $\Re_C$ is used to select clause-instances.*

An X-tree based on a complete X-computation of P via $\Re_A$ and $\Re_C$ is an X-tree of P via $\Re_A$ and $\Re_C$. X-computations and X-trees are (modulo variants) uniquely defined given P, $\Re_A$, and $\Re_C$ because $\Re_A$ and $\Re_C$ are total (pure) functions.

Def 29 *A* fair X-computation *of P is one that uses a fair clause-selection rule. Fair X-trees are based on fair, complete X-computations.*

Example Consider the definite program Pqr2 in Fig 4.5. Each X-tree—including

```
% Program 'Pqr2'.
p <- q, r.
q <- q.
```

Figure 4.5: Definite Program Pqr2.

the fair ones—of Pqr2 is infinite, but some are smaller than the others, merely due to a different atom-selection rule. Fig 4.6 shows an X-tree based on leftmost atom-selection, and Fig 4.7 shows one based on rightmost-atom selection; both the X-trees are based on (fair) FCFS clause-selection.

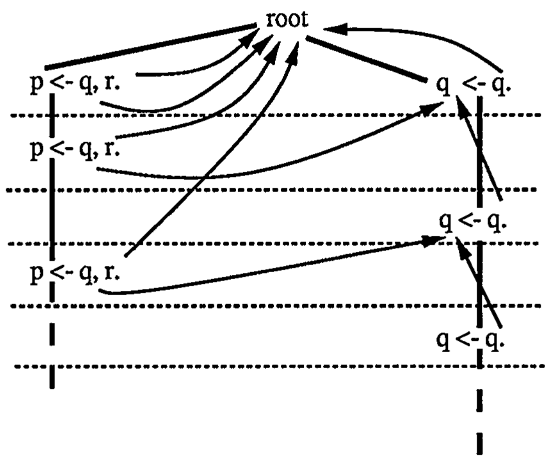


Figure 4.6: A Big X-tree for Pqr2.

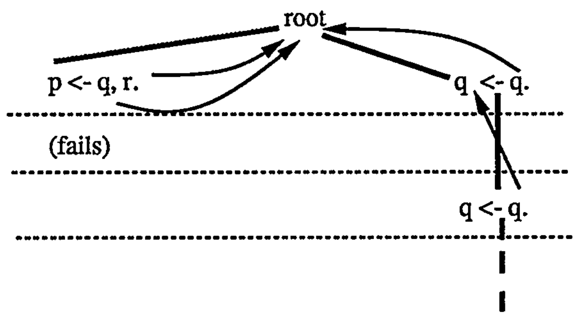


Figure 4.7: A Small X-tree for Pqr2.

Def 30 *Let $\Im$ be some X-tree of P. In $\Im$, an atom $A^{\vec{n}}$, in the body of some clause-instance,* points* in $\Im$ to *a clause-instance Γ iff there exists in $\Im$ an X-derivation from $C\mathcal{L}(\vec{n})$ yielding Γ.*

Theorem 31 *Let $\Gamma = H \leftarrow A_1^{\vec{n}_1}, \ldots, A_q^{\vec{n}_q}$ $(q > 0)$ be a clause-instance occurring in an X-tree $\Im$ of P, and let each A_i point* in $\Im$ to a unit clause-instance Δ_i. For the set $\{[A_1, \ldots, A_q], [\Delta_1+, \ldots, \Delta_q+]\}$, let α be an mgu. Then in $\Im$,*

(a) let Ω be some clause-instance in the X-assertion from $C\mathcal{L}(\vec{n}_m)$ yielding Δ_m, where $1 \leq m \leq q$. (Assume that each Δ_i, Ω, and Γ are suitable variants, i.e., are standardized apart.) Then, in a derivation from Γ yielding Γ', if an atom $A_m^{\vec{n}_m}$ is selected with Ω as the input-clause then there exists an mgu β such that $A_m\beta = (\Omega+)\beta$.

(b) there exists substitution γ such that $\forall i$, $A_i\beta\gamma = A_i\alpha = (\Delta_i+)\alpha = (\Delta_i+)\gamma$ and $H\beta\gamma = H\alpha$.

(c) after such a derivation, each atom in $\Gamma'-$ may be further instantiated with respect to Γ, but only the pointer of A_m can change.

(d) *after such a derivation,* $\forall i,\ i \neq m \rightarrow A_i\beta$ $(\in \Gamma'-)$ *points* in* $\Im$ *to* Δ_i*. If* $\Omega = \Delta_m$ *then* A_m *disappears; otherwise, if* $\mathcal{ID}(\Omega) = r$ *then* $A_m^{\vec{r}}\beta$ *replaces* $A_m^{\vec{n}_m}$ *in* Γ'*, meaning thereby that* $A_m\beta$ *points* to* Δ_m*.*

This means that if the body atoms in a clause-instance Γ point* in $\Im$ to unit clause-instances, and if the sequence of body atoms is unifiable with the sequence of the clause-instances, then there exists a derivation from Γ such that the unifiabilty is retained and the resulting instantiated body atoms continue to point* in $\Im$ to the corresponding unit clause-instances. Also, (b) states that in this derivation, the atoms in Γ never become "over"-instantiated so as to "lose" tuples that can be computed from Γ via some X-assertion. Note that $\Im$ may be an unfair X-tree.

Proof Given Ω is a clause-instance in the X-assertion from $\Pi = \mathcal{CL}(\vec{n}_m)$ yielding Δ_m.

$\Rightarrow$ Δ_m is a clause-instance of Ω (using Def 4).

$\Rightarrow$ There exists an idempotent substitution δ such that $(\Delta_m+) = (\Omega+)\delta$. (As stated in §4.2, we consider only idempotent substitutions.) Let δ be constructed by taking the composition of idempotent mgus used in the X-assertion in $\Im$ from Ω yielding Δ_m. We are given that $A_m\alpha = (\Delta_m+)\alpha$.

$\Rightarrow$ $A_m\alpha = (\Omega+)\delta\alpha = A_m\delta\alpha$ (δ does not act on variables in Γ due to standardization apart).

$\Rightarrow$ A_m and $\Omega+$ are unifiable via $\delta\alpha$.

$\Rightarrow$ There exists an mgu unifying A_m and $\Omega+$.

$\Rightarrow$ There exists an idempotent mgu β such that $A_m\beta = (\Omega+)\beta$, and there exists substitution θ such that $\beta\theta = \delta\alpha$ (using mgu definition in [Llo87, page 23]).

$\Rightarrow$ (a) is true.

Now $\forall i,\ i \neq m \rightarrow A_i\beta\theta = A_i\delta\alpha$. It is implicit that $1 \leq i \leq q$.

$\Rightarrow \forall i,\ i \neq m \rightarrow A_i\beta\theta = A_i\alpha = (\Delta_i+)\alpha = (\Delta_i+)\delta\alpha$ (δ only acts on variables in the X-assertion from Ω yielding Δ_m).

$\Rightarrow \forall i,\ i \neq m \rightarrow A_i\beta\theta = A_i\alpha = (\Delta_i+)\alpha = (\Delta_i+)\beta\theta = (\Delta_i+)\theta$ (β only acts on variables in Ω and Γ).

Also, $(\Delta_m+)\alpha = A_m\alpha = A_m\delta\alpha = A_m\beta\theta = (\Omega+)\delta\alpha = (\Omega+)\delta\delta\alpha$ (δ is idempotent).

$\Rightarrow A_m\beta\theta = (\Delta_m+)\delta\alpha = (\Delta_m+)\beta\theta$.

$\Rightarrow A_m\beta(\beta\theta) = A_m\alpha = (\Delta_m+)\alpha = (\Delta_m+)(\beta\theta)$ (β is idempotent). Combining this with the previous result that $\forall i,\ i \neq m \rightarrow A_i\beta(\beta\theta) = A_i\alpha = (\Delta_i+)\alpha = (\Delta_i+)(\beta\theta)$, we have that there exists substitution γ such that the following holds:

$$\forall i,\ A_i\beta\gamma = A_i\alpha = (\Delta_i+)\alpha = (\Delta_i+)\gamma$$

(by putting $\beta\theta = \gamma$).

Also, $H\alpha = H\delta\alpha = H\beta\theta$ (δ does not act on variables in Γ).

$\Rightarrow H\alpha = H\beta\beta\theta = H\beta\gamma$ (β is idempotent).

$\Rightarrow$ (b) is true.

(c) is true simply because A_m is the only selected atom in the derivation from Γ (using Def 5).

(d) follows from (c) and the given condition that each $A_i \in \Gamma-$ points* in $\Im$ to Δ_i (using Def 5). $\square$

Theorem 32 *Let $\Gamma = H \leftarrow A_1, \ldots, A_q$ $(q \geq 0)$ be a clause-instance of P. Let there be an X-assertion ∂ from Γ using $\Delta_1, \ldots, \Delta_q$ as (suitable variants of) its unit input-clauses and yielding a tuple B. Then, for the set $\{[A_1, \ldots, A_q], [\Delta_1+, \ldots, \Delta_q+]\}$,* *there exists a unifier α such that $B = H\alpha$.*

∂ may use other non-unit input-clauses too. Since there would exist a "minimal" X-assertion that used no other input-clauses other than $\Delta_1, \ldots, \Delta_q$, the Theorem need not and does not argue that α is an mgu.

Proof If $q = 0$ then the Theorem is trivially true because $B = H$. We now consider the case when $q > 0$. Given that for each i, there exists some instance of A_i that disappears, when some clause-instance of Γ is atom dropped on selection with the corresponding Δ_i as unit input-clause (given ∂). Let β_i be the composition of mgus applied in ∂ in sequence to clause-instances of Γ until A_i, or its instance, disappears. (It is not necessary that a body atom be dropped or disappear after each derivation in ∂.)
$\Rightarrow \forall i,\ A_i\beta_i = (\Delta_i+)\beta_i$. (Most of the bindings in β_i would not act on variables in Δ_i since only the last mgu that was composed to result in β_i would act on Δ_i.) Let A_m, or its instance, be the last atom to disappear in ∂. Surely, there exists one such atom for $q > 0$.
$\Rightarrow \forall i,\ A_i\beta_m = (\Delta_i+)\beta_m$ since β_m is the composition of all the mgus applied to (clause-instances of) Γ in ∂ (A_m is last to disappear). Therefore, β_m is a unifier. Since $B = (\Gamma+)\beta_m = H\beta_m$, we have the proof by replacing β_m by α. $\Box$

Theorem 33 *Let $\Gamma = H \leftarrow A_1, \ldots, A_q$ $(q \geq 0)$ be a clause in P, and let there be an X-assertion from Γ using $\Delta_1, \ldots, \Delta_q$ as its unit input-clauses and yielding a tuple B. Let $\Im$ be the X-tree of P via an atom-selection rule $\Re_A$ and a fair clause-selection rule $\Re_C$. If each $A_i \in \Gamma-$ points* in $\Im$ to a unit clause-instance Δ'_i, then in $\Im$, there exists an X-assertion ∂ from Γ using $\Delta'_1, \ldots, \Delta'_q$ as its unit input-clauses and* *yielding a tuple B'. Here, each Δ'_i is a suitable variant of Δ_i, and B' is a suitable variant of B.*

∂ and the X-assertion from Γ yielding the tuple B may use other non-unit input-clauses too. This Theorem states that if a tuple B is yielded by some X-assertion—not necessarily in some X-computation—from a clause in P and the unit clause-instances used by the X-assertion are X-asserted in a fair X-tree $\Im$ of P, then a variant B' is yielded in $\Im$.

Proof Given an X-assertion from Γ using $\Delta_1, \ldots, \Delta_q$ as unit input-clauses and yielding a tuple B.

$\Rightarrow$ There exists a unifier α for the set $\{[A_1, \ldots, A_q], [\Delta_1+, \ldots, \Delta_q+]\}$ such that $B = H\alpha$ (using Theorem 32).

$\Rightarrow$ There exists a unifier α' for the set $\{[A_1, \ldots, A_q], [\Delta'_1+, \ldots, \Delta'_q+]\}$ such that $B' = H\alpha'$ (each Δ'_i is a suitable variant of Δ_i, and B' is a suitable variant of B). Given that each Δ'_i occurs in $\Im$.

$\Rightarrow$ In $\Im$, there are X-assertions from clauses in P yielding each Δ'_i (Δ'_i is a unit clause-instance).

$\Rightarrow$ Each $A_i \in \Gamma-$ points* in $\Im$ to the corresponding Δ'_i (each A_i is $A_i^{\vec{0}}$ in *init*).

$\Rightarrow$ In $\Im$, there exists an X-derivation ∂ from Γ such that in each of the clause-instances comprising ∂:

- the sequence of body atoms is unifiable, via α', with the corresponding unit clause-instances, and

- each A_i, or its instance, either has disappeared on selection with Δ'_i as unit input-clause or points* in $\Im$ to Δ'_i (using Theorem 31).

Successive clause-instances in the X-assertions yielding each Δ_i' are being yielded as the X-computation of P via $\Re_A$ and $\Re_C$ proceeds (X-assertions for each Δ_i' exist in $\Im$).

$\Rightarrow$ Each Δ_i' is eventually X-asserted in a complete X-computation via $\Re_A$ and $\Re_C$. Each clause-instance in ∂ is eventually—and in sequence—X-executed ($\Re_C$ is fair). This forces the eventual selection of each Δ_i' as input-clause with A_i, or its instance, as the selected atom (using Def 10). Hence, each A_i, or its instance, eventually disappears on selection with Δ_i' as unit input-clause ($\Re_C$ is fair, Def 10).

$\Rightarrow$ Tuple $H\alpha' = B'$ is eventually yielded. □

Def 34 *Let $\Re_A$ be an atom-selection rule and $\Re_C$ be a clause-selection rule. An atom A is an $\Re_{AC}$-*computed tuple *iff there exists an X-computation $\mho$ via $\Re_A$ and $\Re_C$ of P such that the unit clause-instance $A \leftarrow$ occurs in some stage of $\mho$.*

Every X-computation is carried out by some clause-selection and atom-selection rules. We try to mention the rules only when necessary.

Def 35 *Let $\Re_A$ be an atom-selection rule and $\Re_C$ be a clause-selection rule. The* $\Re_{AC}$-success set *$SS_{P:AC}$ is the set of all $A \in B_P$ such that $A = A'\alpha$, for some $\Re_{AC}$-computed tuple A' and (ground) substitution α.*

Unlike SS_P (see Def 13), $SS_{P:AC}$ contains grounded tuples computed by exactly one X-computation (modulo variants), the one via $\Re_A$ and $\Re_C$.

Theorem 36 ($SS_{P:AC} = M_P$**, Completeness of Search)** *Let $\Re_A$ be an atom-selection rule and $\Re_C$ be a fair clause-selection rule. The $\Re_{AC}$-*success set *of a definite program is equal to its least, Herbrand model.*

Given a correct ground tuple A, we show that there exists a corresponding (possibly non-ground) $\Re_{AC}$-computed tuple, which is X-asserted finitely. However, this is not to say that tuples corresponding to all (ground) correct tuples will be computed in a predetermined, finite number of steps. Also, some X-trees may not have computed tuples corresponding to some of the correct tuples, but overall, corresponding to each correct tuple, a computed tuple will appear on at least one X-tree. What does this mean in practice? Do all the X-trees have to be constructed and the computed tuples collected from each of them to build a program's minimal model? Or, is there some magical, nondeterministic manner in which a "correct" X-tree can be constructed? We dispel the magic and prove that any fair X-tree can be selected for construction, guaranteed that it will be "correct." This result establishes the completeness of fair X-computations.

Proof Let P be a definite program, SS_P be its success set, $SS_{P:AC}$ be its $\Re_{AC}$-success set, M_P be its least, Herbrand model, and $\Im$ be the X-tree of P via $\Re_A$ and $\Re_C$. Since $SS_{P:AC} \subseteq SS_P$, it suffices TPT M_P is contained in $SS_{P:AC}$ (using Corollary 19).

$\Rightarrow$ TPT $\forall A,\ A \in M_P \rightarrow A \in SS_{P:AC}$. We are given that P is definite.

$\Rightarrow$ TPT $\forall A,\ A \in T_P{\uparrow}\omega \rightarrow A \in SS_{P:AC}$ (using Theorem 6.5 in [Llo87, page 38]). Now, $\forall A\, \exists n\ (n \in \omega),\ A \in T_P{\uparrow}\omega \leftrightarrow A \in T_P{\uparrow}n$.

$\Rightarrow$ TPT $\forall A,\ A \in T_P{\uparrow}n$ for some $n \in \omega \rightarrow A \in SS_{P:AC}$ (using Theorem 6.5 in [Llo87, page 38]).

$\Rightarrow$ TPT $\forall A,\ A \in T_P{\uparrow}n$ for some $n \in \omega \rightarrow$ there exists $\Re_{AC}$-computed tuple A', which occurs in $\Im$, such that $A = A'\alpha$, for some ground substitution α (using Def 35). We induce on n to prove this implication. Let LHS be the proposition that $A \in T_P{\uparrow}n$

and RHS be the proposition that there exists an $\Re_{AC}$-computed tuple A' such that $A = A'\alpha$, for some ground substitution α.

Basis: For $n = 0$, the statement LHS $\rightarrow$ RHS is vacuously true since $T_P{\uparrow}0 = \{\}$. Just as an exercise, we try TPT LHS $\rightarrow$ RHS for $n = 1$. Given $A \in T_P{\uparrow}1$.

$\Rightarrow$ There exists a clause $A' \leftarrow$ in P such that $A = A'\alpha$, for some ground substitution α.

$\Rightarrow$ There exists an X-assertion of unit length from $A' \leftarrow$ yielding itself (using Def 6). This X-assertion is carried out by each X-computation of P (using Def 11).

$\Rightarrow$ RHS (using Def 12).

Hypothesis: Assume that for $1 < n \leq k - 1$ $(k > 1)$, LHS $\rightarrow$ RHS.

Induction: TPT for $n = k$, LHS $\rightarrow$ RHS. Given that $A \in T_P{\uparrow}k$.

$\Rightarrow$ There exists a clause $\Gamma = B \leftarrow B_1, \ldots, B_q$ in P such that for some ground substitution α, $A = B\alpha$ and $\{B_1\alpha, \ldots, B_q\alpha\}$ is contained in $T_P{\uparrow}(k-1)$ (using definition of T_P in [Llo87, page 37].

$\Rightarrow$ There exist $\Re_{AC}$-computed tuples $C_1, \ldots, C_q$ such that for some ground substitution β, $\forall i$ $(1 \leq i \leq q)$, $B_i\alpha = C_i\beta$ (using Hypothesis). These tuples are surely X-asserted in $\Im$.

$\Rightarrow$ In $\Im$, each atom B_i in $\Gamma-$ points* to a unit clause-instance $C_i \leftarrow$. This is because each body atom of clause-instances in *init* refers to $\vec{0}$.

Now, $\forall i$ $(1 \leq i \leq q)$, there exists an X-assertion from some clause in P ending in the unit clause $C_i \leftarrow$ (using Def 12). Each clause-instance $C_i \leftarrow$ may be used as an input-clause in a derivation from a clause-instance in *init*. Also, each body atom B_i is unifiable with C_i, e.g., by the unifier $\alpha\beta$, which means that $[B_1, \ldots, B_q]$ is unifiable with $[C_1, \ldots, C_q]$, say, via mgu γ.

$\Rightarrow \alpha\beta = \gamma\theta$, for some (ground) substitution θ.

$\Rightarrow$ There exists an X-assertion, possibly outside $\Im$, from Γ such that each body atom B_i is selected in turn with $C_i \leftarrow$ as input-clause. Let this X-assertion yield a tuple A', which would be a variant of $B\gamma$. Surely, $A = B\alpha = B\alpha\beta = B\gamma\theta = A'\delta$, for some ground substitution δ.

$\Rightarrow$ In $\Im$, a tuple A'' is X-asserted such that A'' is a variant of A' (using Theorem 33).

$\Rightarrow$ RHS is true (using Def 12). $\square$

As shown by Theorem 36, given a definite program, the policy of atom selection does not affect an X-computation's soundness and completeness. (We had proved an atom-switching lemma stating that switching selected atoms in an X-assertion gives another X-assertion such that the tuples yielded by the two X-assertions are variants. Since this lemma was not required for proving completeness, we did not include it in this thesis.) This means that we can arbitrarily fix an atom-selection rule before beginning to execute a definite program. This is not true when negation is allowed and normal programs are considered because the *finite* failure of X-derivations becomes important.

As described in Chapter 3, our Starlog interpreter uses all-atom selection. In all-atom selection, all the atoms in the body of the clause-instance are selected in some arbitrary sequence and a descendant is computed. All-atom selection shrinks paths in X-trees and thereby may cause "early" failures and successes. All-atom selection ensures fairness towards atoms and greater interleaving in the search, which resembles a breadth-first search. Thus, we circumvent the problem of fairness towards atoms *within* a clause-instance. This is a requirement stricter than required for guaranteeing completeness of search in case of normal programs.

Theorem 37 *Let $\Gamma = H \leftarrow A_1, \ldots, A_q$ $(q \geq 0)$ and $\Delta = F \leftarrow B_1, \ldots, B_r$ $(r \geq 0)$ be clauses in P that have been standardized apart. Let $\Im$ be the X-tree of P via an atom-selection rule $\Re_A$ and a fair clause-selection rule $\Re_C$.*

(a) suppose, in an X-tree $\Im_a$, C is a tuple X-asserted from Δ in depth n using $\Sigma_1, \ldots, \Sigma_r$ as its unit input-clauses. Then in $\Im$, there exists an X-assertion from Δ yielding C' using $\Sigma'_1, \ldots, \Sigma'_r$ as its unit input-clauses. Here, C and $\Sigma_1, \ldots, \Sigma_r$ are suitable variants of C' and $\Sigma'_1, \ldots, \Sigma'_r$.

(b) suppose, in an X-tree $\Im_b$, D is a tuple X-asserted from Γ in depth $n+1$ using $\Xi_1, \ldots, \Xi_q$ as its unit input-clauses. Then, each atom $A_i \in \Gamma-$ points in $\Im$ to Ξ'_i. Each Ξ'_i is a suitable variant of Ξ_i.*

Both C and D may be X-asserted using other non-unit input-clauses, in addition. A computed tuple A of P, which is X-asserted in some X-tree of P, is also X-asserted in each fair X-tree $\Im$ of P.

This Theorem is not referred to by any other theorem in this chapter, but it is an important result and can aid future improvements on or extensions to Starlog's procedural semantics.

Proof We induce on the depth n.

Basis: $n = 0$.

$\Rightarrow$ In $\Im_a$, length of X-assertion yielding $C \leftarrow$ is 1 (using Def 16).

$\Rightarrow$ $\Delta = F \leftarrow = C \leftarrow$ is a unit clause in P.

$\Rightarrow$ In $\Im$, there exists an X-assertion from Δ of unit length yielding $C \leftarrow$. There are no input-clauses in this X-derivation.

$\Rightarrow$ Basis for (a).

In $\Im_b$, D is X-asserted from Γ in depth $n+1=1$.

$\Rightarrow$ Γ has only one atom A_1 in its body, and in the X-assertion in $\Im_b$ from Γ yielding D, a unit clause Ξ_1 in P is used as the input-clause (using Def 16). Being a clause in P, Ξ_1 is X-assertible in depth 0 in every X-tree of P. Now in $\Im$, body atoms in clause-instances in *init* point to *root*.

$\Rightarrow$ A_1 points* in $\Im$ to each clause in P since each clause in P is assumed X-derivable (in $\Im$) from *root*.

$\Rightarrow$ A_1 points* in $\Im$ to Ξ_1.

$\Rightarrow$ Basis for (b).

Hypothesis: Assume that for each n such that $0 < n \leq k$ $(k > 0)$, the Theorem is true.

Induction: $n = k+1$. Given that in $\Im_a$, $C \leftarrow$ is X-assertible from Δ in depth $k+1$ using $\Sigma_1, \ldots, \Sigma_r$ as the unit input-clauses.

$\Rightarrow$ Each Σ_i is X-assertible in $\Im_a$ from some clause in P in a depth $\leq k$ (using Def 16).

$\Rightarrow$ In $\Im$, each $B_i \in \Delta-$ points* in $\Im$ to Σ'_i (using Hypothesis (b)), and there exist X-assertions yielding each Σ'_i (using Hypothesis (a)).

$\Rightarrow$ In $\Im$, there exists an X-assertion from Δ yielding C', where C' is a suitable variant of C (using Theorem 33).

$\Rightarrow$ Hence, the proof for (a).

Given $\Gamma = H \leftarrow A_1, \ldots, A_q$ $(q \geq 0)$, and that D is a tuple X-asserted in $\Im_b$ from Γ in depth $n+1 = k+2$ using $\Xi_1, \ldots, \Xi_q$ as unit input-clauses.

$\Rightarrow$ In $\Im_b$, each Ξ_i is X-asserted in depth $\leq k+1$ (using Def 16) from a clause Ω_i in P using unit input-clauses, say, $\Lambda_1, \ldots$

$\Rightarrow$ Each atom $E_j \in \Omega_i-$ points* in $\Im$ to Λ'_j (using Hypothesis (b)). Each Λ'_j is a suitable variant of Λ_j.

$\Rightarrow$ In $\Im$, for each Ξ'_i, there exists an X-assertion from Ω_i yielding it (using Theorem 33 on each Ξ'_i).

In $\Im$, each body atom in *init* refers to *root*.

$\Rightarrow$ Each $A_i \in \Gamma-$ points* in $\Im$ to each and every clause, including Ω_i, in P.

$\Rightarrow$ Each $A_i \in \Gamma-$ points* in $\Im$ to Ξ'_i (using Def 30). Hence, the proof for (b). □

In this chapter, we have formally defined definite-Starlog execution, via fair X-computations, and have proved both its soundness and completeness with respect to least, Herbrand models. Although inspired by the proofs for SLD-resolution in [Llo87], the proofs here are novel and tackle more-complicated problems:

- our results are for Starlog's X-computations, which employ a bottom-up, model-theoretic form of deduction; the results in [Llo87] are for top-down SLD-resolution.

- X-computations may use as input-clauses clause-instances that are not part of the original program, but are deduced from it. Therefore, correctness proofs are more complicated for X-computations than for SLD-resolution, which uses only clauses from the given program as input-clauses.

- according to Lloyd [Llo87, page 56], while any two SLD-trees may have greatly different size and structure, they are essentially the same with respect to success branches. SLD-trees differ in their selection of atoms from a query or goal. An X-tree for a Starlog program is built according to a clause-selection rule, and there can be an X-assertion (success path) in one for which there is no equivalent in another. Except for fair X-trees, X-trees are not alike with respect to success paths.

- in an SLD-tree, goals deduced along one path from a node (goal) do not affect goals deduced along a different path from that node. Therefore, although Lloyd [Llo87] does not present an explicit proof of completeness of search, i.e., for finding success branches in an SLD-tree, such a proof would be trivial—when considering fair clause-selection—given the strong completeness of SLD-resolution [Llo87, Theorem 9.5]. In contrast, in an X-tree, a clause-instance deduced along one path from a node (clause-instance) may be used as an input-clause when deducing along a different path from that node. Due to this interconnection between paths, our results leading to a proof of completeness of search, Theorem 36, are complicated.

We now move on to Chapter 5, which describes Starlog negation.

Chapter 5

Negation in Starlog

Starlog uses negation and constraints on explicit time to model mutation and persistence. Therefore, improved, correct handling of negation will improve Starlog's promise of an effective, declarative paradigm for updates. Starlog negation has to be efficient, and negations should be satisfied or failed quickly.

Starlog negation is based on the completed program and equality axioms, as in [Llo87, §14], but negation via SLD with Negation as Failure (SLDNF) [Llo87] is unsatisfactory. In plain or *unsafe* SLDNF, a negated literal can be selected for execution even if there are non-ground, non-local variables within it, i.e., there is no floundering or delaying for a safe condition. (Non-local variables are those that are existentially or universally quantified outside the negation.) Unsafe SLDNF is unsound with respect to Herbrand models [Llo87, §15]; so, it cannot be used for Starlog. *Safe* SLDNF flounders (delays) a negated literal until each non-local variable in the negation's body is ground. Therefore, it is sound, but being a test, it does not permit a negation to generate bindings for non-local variables. Therefore, generate-and-test programs can be inefficient and might never terminate. Consider the program in Fig 5.1. With SLDNF, the constraints on the bindings of variables occurring within a negation are not exported outside the negation. Therefore, given a goal `p(N)`, the interpreter would test the negation on each of the hundred, possible integers `N` given by `num(N)`. This is instead of using the negation to deduce the rule-instance `p(N~(-inf,3]) <- num(N~(-inf,3])`, which constrains the search

```
% Program Gen-Test.
% 'num' generates integers lying between 0 and 100,
% and 'p' tests whether they are less than or equal to 3.

p(N) <- num(N), not(N > 3).
num(0).
num(N) <- num(Nprev), Nprev >= 1, +(Nprev,1,N), 100 > N.
```

Figure 5.1: A Generate-and-Test Program.

for num tuples.

For logic programming—as opposed to theorem proving, which checks for consistency—it is desirable to use negations as constructors or generators of bindings rather than as mere tests. Negations should be used to constrain the search space.

Negation in Starlog is not based on passive SLDNF, as it is in Prolog [CM81]. Instead, it is constructive. A negated literal can be selected at any stage for execution and can generate bindings for all of its variables—just like any other constraint.

This chapter does not attempt to prove that Starlog negation is sound and complete. (Some intuition on how to approach such a proof is given in §7.3.) It only presents the motivation for temporal stratification and describes the rewrites and general strategy used to implement negation in Starlog. We use "extended, Herbrand model" to mean a Herbrand model that includes equality axioms and the set of real numbers as implicit constants in a given program. We do not enumerate the equality axioms, though, when writing out such a model.

5.1 Temporal Stratification

A program is said to be *consistent* if it has a model. Every *normal* program, which allows negated literals in the bodies of its rules, is consistent, but its completion may not be consistent [Llo87, page 83]. Temporal stratification, mentioned in Chapter 2, is a syntactic condition sufficient to ensure that the completion of a normal program is consistent. Its motivation, as for ordinary stratification [Llo87, page 83], is to limit the use of negation in recursive rules to keep the model theory manageable.

As mentioned in Chapter 2, Starlog restricts its domain to a class of temporally-stratified programs. This ensures that when negation is used, Starlog's bottom-up execution proceeds monotonically, i.e., tuples once yielded do not have to be deleted or undone. Recall that a sufficient condition for a Starlog program to be temporally stratified is that each recursive, predicate-call loop involving negation should be accompanied by a time advance, i.e., there should be no zero-delay loops involving negation [CK91]. (This condition can be checked for during execution.) Clearly, this class of temporally-stratified programs is a superset of the class of programs stratified, as per [Llo87, page 83], on predicate symbols alone.

5.2 Type Mismatch: Error or Failure?

Before we produce the rewrites for negation, we digress here to point out the question of incorrect types of arguments to language primitives. If a primitive expects an argument of some type and receives one of a different type, then whether the primitive should fail silently or raise an error or exception is a controversial question. Consider the following program:

```
    p(X) <- not(X > 5).
    r(a).
```

The extended, Herbrand model {`r(a), p(a), p((-inf,5])`} is the intended model according to the view of "silent failure on type mismatch," which requires that literal `(a > 5)` fail silently. The other view seeks {`r(a), p((-inf,5])`} as the intended model of the program. Although important, this controversy is beyond the scope of this thesis, and we shall not discuss it any further. Suffice to say that the rewrites in this chapter take the view of "silent failure on type mismatch."

5.3 Negation Strategy

As mentioned in Chapter 3, Starlog selects from its scheduling list a rule-instance, rewrites it, deduces new rule-instances or tuples from it in the X-tree, rewrites the resultant rule-instances, and places the non-tuples amongst them in the scheduling list for future execution. We now describe the forward and return rewrites.

5.3.1 Forward Rewrites

The period after a rule-instance's selection from the scheduling list and before deduction in the X-tree is called the *forward phase.* In our Starlog interpreter, some rewrites are performed during a forward phase; these are called *forward rewrites.* Fig 5.2 lists the forward rewrites, which are applied repeatedly on the rule-instance until they can cause no further change. `not(A, B, ...)` is rewritten into the more-general form `not-exists: (A, B, ...)` and then executed.

In the forward phase, the '`;`' call is "eagerly" expanded to yield as many rule-instances as there are literals in its argument, and then, each of those rule-instances is executed in some order. The '`;`' expansion is "eager" because a possible failure

#	Argument	Forward Rewrite
F1	p(U) <- q(V), (r(W); s(X); ...), t(Y).	p(Uw) <- q(Vw), r(Ww), t(Yw). p(Ux) <- q(Vx), s(Xx), t(Yx). p(Uz) <- q(Vz), ..., t(Yz).
F2	X =/= Y	not(X = Y)
F3	not(A, B, ...)	not-exists: (A, B, ...)

Figure 5.2: Forward Rewrites.

of other literals in the rule-instance's body is not awaited. On the other hand, by "delaying" the '`;`' expansion of a rule-instance R till the forward phase—instead of expanding R at the end of the earlier execution that generated R—branching in the X-tree is delayed till it is clear that the expanded rule-instances will not trivially fail.

5.3.2 Return Rewrites

After the forward rewrites have been repeatedly applied, a rule-instance is executed using Starlog's bottom-up deduction. This may result in some tuples and some non-tuple rule-instances. Now, the return phase follows. In the *return phase*, the resultant, non-tuple rule-instances are subjected to rewrites, called *return rewrites*, before they are placed into the scheduling list for future execution. Like their forward counterparts, the return rewrites too are applied repeatedly until they can cause no further change. They are also applied right at the beginning of Starlog execution, when the program's rules are being placed into the scheduling list. We now describe these rewrites under three categories: those for inversion, those for (back-) converting `not-exists` literals into the more-specialized `not` literals, and those for partitioning.

Rewrites for Inversion

Fig 5.3 summarizes the return rewrites used for inverting the arithmetic primitives when the arguments are known to be timestamps. Recall from Chapter 2 that timestamps occur as the first arguments in user-defined predicates and are real valued.

#	Literal	Rewrite
R1	not(T > U)	U >= T
R2	not(T >= U)	U > T
R3	not(T != U)	T =:= U
R4	not(T =:= U)	U > T; T > U
R5	not(real(T))	false
R6	not(notreal(T))	true

Figure 5.3: Rewrites for Inversion of Timestamp Constraints.

Although most arithmetic constraints in Starlog programs would involve timestamps, we need to cater to general arguments too. Fig 5.4 summarizes the general rewrites used for inverting arithmetic primitives. (Refer to Chapter 2 for a list of Starlog's primitives.) Fig 5.4 generalizes Fig 5.3 in that it additionally handles arguments that are not real valued and yet occur in arithmetic constraints. Rewrite R4, in both Fig 5.3 and Fig 5.4, prefers (`X > Y; Y > X`) over (`X != Y`), although these rewrites are equivalent; we disclose the reason for this only later in §7.2.2.

#	Literal	Rewrite
R1	not(X > Y)	notreal(X); notreal(Y); Y >= X
R2	not(X >= Y)	notreal(X); notreal(Y); Y > X
R3	not(X != Y)	notreal(X); notreal(Y); X =:= Y
R4	not(X =:= Y)	notreal(X); notreal(Y); Y > X; X > Y
R5	not(real(X))	notreal(X)
R6	not(notreal(X))	real(X)

Figure 5.4: Rewrites for General Inversion of Arithmetic.

#	Argument Literal	Rewrite
R7	not(X = Y)	X =/= Y
R8	not(not(A, B, ...))	A, B, ...
R9	not(not-exists X,Y,...: (A, B, ...))	A, B, ...
R10	not(A, B, C, ...)	not(A); (A, not(B)); (A, B, not(C)); ...
R11	not(A; B; ...)	not(A), not(B), ...

Figure 5.5: Remaining Rewrites for Inversion.

Fig 5.5 lists the remaining rewrites for inversion. Rewrite R9 of Fig 5.5, as formulated, may be used only at the top level of a rule-instance so that correctness is not sacrificed. Rewrite R10 of Fig 5.5 is done only when each positive literal, or atom, within `(A, B, ...)` is a primitive that is incapable of forcing solutions on its own, i.e., whose repeated execution does not cause further instantiations or "squeezing" of real intervals associated with arithmetic variables. Since the disjuncts rewritten into end up in different rule-instances in the X-tree—as explained earlier—this "delay" in using Rewrite R10 of Fig 5.5 serves to reduce the branching in the X-tree.

Rewriting `not-exists` as `not`

This is an important, return rewrite that drops variables local to a `not-exists` literal; a `not` literal results if all such variables can be dropped. The `not` literal is more active since it can invert its body; hence the importance of this rewrite.

Fig 5.6 shows some examples where local, existentially-quantified variables can be dropped from within a `not-exists` literal. A local, existentially-quantified variable can be dropped when it satisfies any of the following conditions:

- it is fully instantiated.

Example	Rewrite
not-exists X~a: (r(X~a))	not(r(a))
not-exists X: (r(Y), p(Z))	not(r(Y), p(Z))
not-exists X~f(Z,b),Y~[1,3): (r(Y~[1,3)), p(X~f(Z,b)))	not-exists Y~[1,3): (r(Y~[1,3), p(f(Z,b)))

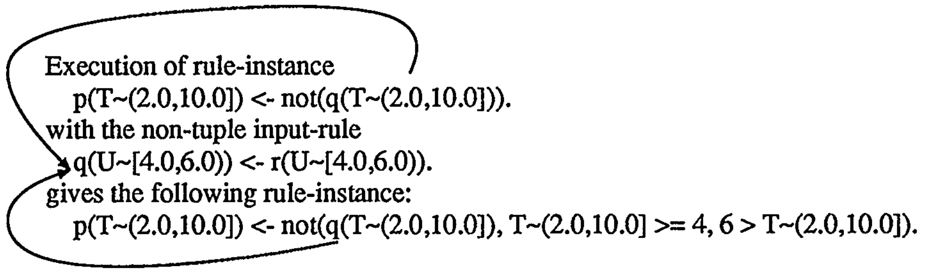
Figure 5.6: Rewriting `not-exists` As `not`: Examples.

- it does not occur in the body of the negation.
- it has been bound to a structure whose variables are universally quantified.

A `not-exists` call is specialized into or rewritten as a call to a `not` if it does not (existentially) quantify any variables within it.

Partitioning

Now, we discuss a return rewrite called partitioning that reasons about arithmetic intervals to quicken the satisfaction of negations. Consider the execution trace in Fig 5.7. Left alone, the rule-instance for q would have to fail or bear tuples before



```
Execution of rule-instance
   p(T~(2.0,10.0]) <- not(q(T~(2.0,10.0])).
with the non-tuple input-rule
   q(U~[4.0,6.0)) <- r(U~[4.0,6.0)).
gives the following rule-instance:
   p(T~(2.0,10.0]) <- not(q(T~(2.0,10.0]), T~(2.0,10.0] >= 4, 6 > T~(2.0,10.0]).
```

Figure 5.7: Execution Trace to Motivate Partitioning.

any tuples of p will be yielded. To improve the efficiency of negation, we look at the arithmetic intervals constraining T inside and outside the negation—this is depicted in Fig 5.8. Clearly, the interval constraining T within a negation can only

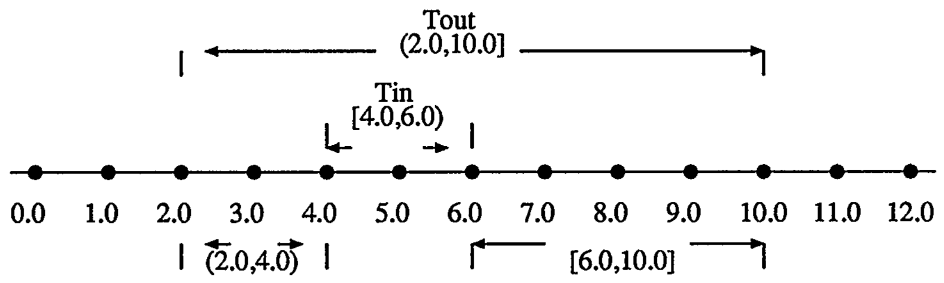
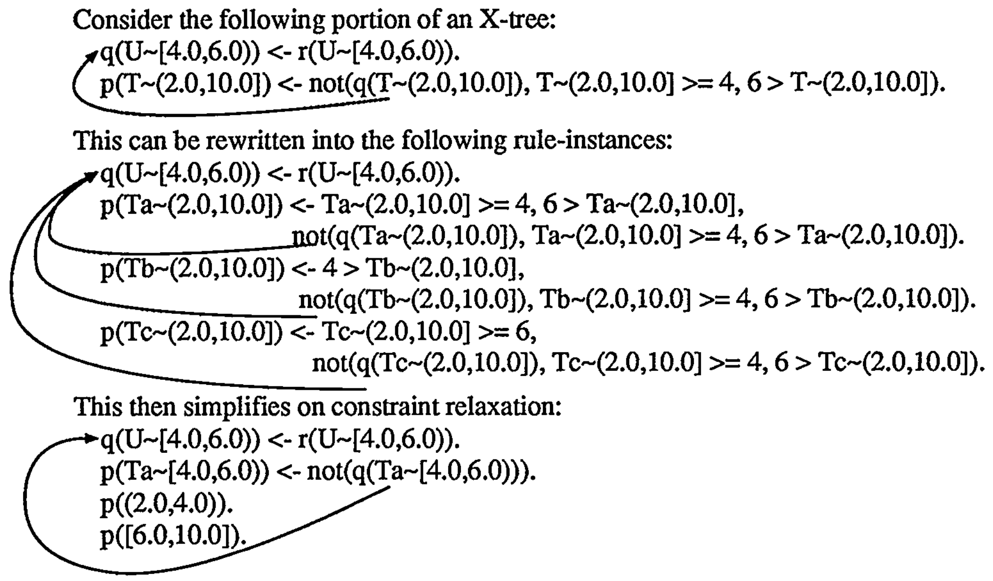


Figure 5.8: Inter-Relationship of Constraining Intervals.

lie within the interval constraining T outside the negation. So, if `Tin` and `Tout` are the respective intervals constraining `T` within and outside the negation, then `Tin` has to lie within `Tout`. This means that `Tout` can be split into three, disjoint intervals, one of which is `Tin`—from Fig 5.8, `Tout~(2.0,10.0]` can be split into `(2.0,4.0)`, `Tin~[4.0,6.0)`, and `[6.0,10.0]`. Therefore, the result of Fig 5.7 can be rewritten into three rule-instances as per Fig 5.9. The tuples `p((2.0,4.0))`



```
Consider the following portion of an X-tree:
q(U~[4.0,6.0)) <- r(U~[4.0,6.0)).
 p(T~(2.0,10.0]) <- not(q(T~(2.0,10.0]), T~(2.0,10.0] >= 4, 6 > T~(2.0,10.0]).

This can be rewritten into the following rule-instances:
q(U~[4.0,6.0)) <- r(U~[4.0,6.0)).
 p(Ta~(2.0,10.0]) <- Ta~(2.0,10.0] >= 4, 6 > Ta~(2.0,10.0],
                     not(q(Ta~(2.0,10.0]), Ta~(2.0,10.0] >= 4, 6 > Ta~(2.0,10.0]).
 p(Tb~(2.0,10.0]) <- 4 > Tb~(2.0,10.0],
                     not(q(Tb~(2.0,10.0]), Tb~(2.0,10.0] >= 4, 6 > Tb~(2.0,10.0]).
 p(Tc~(2.0,10.0]) <- Tc~(2.0,10.0] >= 6,
                     not(q(Tc~(2.0,10.0]), Tc~(2.0,10.0] >= 4, 6 > Tc~(2.0,10.0]).

This then simplifies on constraint relaxation:
q(U~[4.0,6.0)) <- r(U~[4.0,6.0)).
 p(Ta~[4.0,6.0)) <- not(q(Ta~[4.0,6.0))).
 p((2.0,4.0)).
 p([6.0,10.0]).
```

Figure 5.9: Rewrites Via Partitioning.

and `p([6.0,10.0])` have thus been forced out quickly. Such a rewrite is called *partitioning.* Partitioning takes a rule-instance that has negations in its body, selects a real-valued variable X that appears within a negation and is quantified outside it, and rewrites it as different—in this case, three—rule-instances by creating disjoint partitions of the interval of real values in which X lies. In this way, partitioning extends the logical, interval arithmetic of [Cle87] into the domain of negation.

Partitioning is applicable in the presence of both `not-exists` and `not` literals. When there are multiple `not-exists` or `not` literals in the body of a rule-instance, the timestamp T in the head may be partitioned with respect to various possible arithmetic intervals:

- T's interval in the "first" negation that (non-trivially) constrains T.
- T's interval in the negation that constrains T the "most."
- all of T's intervals in the negations that constrain T, after optimizing on the overlap of intervals.

Presently, we use the first option in our interpreter, and restrict partitioning to the timestamp in the heads of rule-instances. Partitioning can be extended to other universally-quantified, real-valued variables occurring in the rule-instance, in the head or in the body.

5.4 Execution of `not-exists`

5.4.1 Example

Let us first consider an execution of a `not-exists` literal. Fig 5.10 depicts a portion of some X-tree.

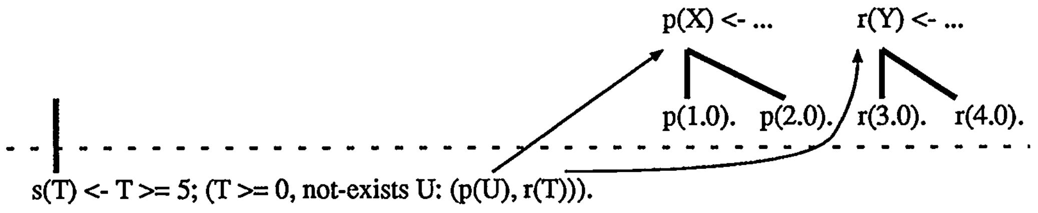


Figure 5.10: A Portion of an X-tree.

Consider the execution of following rule-instance of Fig 5.10 in some detail:

```
s(T) <- T >= 5; (T >= 0, not-exists U: (p(U), r(T))).
```

Assume that Starlog has selected it from its scheduling list.

First, the forward rewrites are applied. Rewrite F1 of Fig 5.2 is applied once to give the following rule-instances:

```
s(Tt) <- Tt >= 5.
s(Tn) <- (Tn >= 0, not-exists Un:  (p(Un), r(Tn))).
```

Forward rewrites do not affect the first of these rule-instances, which is then executed to give the tuple:

```
s(Tt~[5.0,+inf)).
```

Forward rewrites do not affect the second of these rule-instances too. So, this rule-instance's execution in the X-tree begins. Its leftmost literal is selected. This constraint calls Starlog's >= built-in, which binds `Tn` to the interval `[0.0,+inf)`. Next, we select the lone `not-exists` literal in the rule-instance:

```
s(Tn~[0.0,+inf)) <- not-exists Un:  (p(Un), r(Tn~[0.0,+inf))).
```

This causes the negation's body to be executed as though it were a new rule-instance:

```
h <- p(Un), r(Tn~[0.0,+inf)).
```

(The h tuples are treated specially—they are not placed on the X-tree.) Four equalities, corresponding to four possible solutions, result from this execution:

```
Un =:= 1, Tn~[0.0,+inf) =:= 3
Un =:= 1, Tn~[0.0,+inf) =:= 4
Un =:= 2, Tn~[0.0,+inf) =:= 3
Un =:= 2, Tn~[0.0,+inf) =:= 4
```

We take these equalities back to the negation, whose body was being executed, and since Un is existentially quantified within the negation, we have the following result after deduction:

```
s(Tn) <- not-exists Ua~1.0:  (Tn~[0.0,+inf) =:= 3),
         not-exists Ub~1.0:  (Tn~[0.0,+inf) =:= 4),
         not-exists Uc~2.0:  (Tn~[0.0,+inf) =:= 3),
         not-exists Ud~2.0:  (Tn~[0.0,+inf) =:= 4).
```

Now comes the turn for the return rewrites to be applied. As exemplified in Fig 5.6, each of the not-exists conjuncts is specialized into not literals to give the following rule-instance:

```
s(Tn) <- not(Tn~[0.0,+inf) =:= 3), not(Tn~[0.0,+inf) =:= 4),
         not(Tn~[0.0,+inf) =:= 3), not(Tn~[0.0,+inf) =:= 4).
```

Rewrite R4 of Fig 5.3 changes this into the following rule-instance:

```
s(Tn~[0.0,+inf)) <- (Tn~[0.0,+inf) > 3; 3 > Tn~[0.0,+inf)),
                                (Tn~[0.0,+inf) > 4; 4 > Tn~[0.0,+inf)),
                                (Tn~[0.0,+inf) > 3; 3 > Tn~[0.0,+inf)),
                                (Tn~[0.0,+inf) > 4; 4 > Tn~[0.0,+inf)).
```

Even though its body has only arithmetic primitives, this rule-instance is not a tuple and is placed into the scheduling list for future execution; unfortunately, our rewrites are not "intelligent" enough to realize that arithmetic constraints have been duplicated in the body. In the X-tree, this rule-instance and the previously-computed tuple `s(Tt~[5.0,+inf))` are made children rule-instances of the rule-instance we began execution with:

```
    s(T) <- T >= 5; (T >= 0, not-exists U: (p(U), r(T))).
```

On future selection—guaranteed with a fair scheduling—of the resultant non-tuple rule-instance, the '`;`' calls are expanded using the forward rewrite F1 of Fig 5.2 to give sixteen, new rule-instances! Only three of these finally succeed to give the following tuples:

```
    s((0.0,3.0)).
    s((3.0,4.0)).
    s((4.0,+inf)).
```

5.4.2 General Execution

In §5.4.1, we have traced through an example execution of a `not-exists` literal; here, we try to generalize the execution.

Each (partial) execution of a negation's body can potentially result in a conjunction of `not-exists` literals, and these conjuncts have their existentially-quantified variables standardized apart from one another. In more formal terms, suppose a negation N, say, `not-exists E1,...,Em: (B)`, is encountered. Let the body B be a sequence of literals involving a list of variables U that—unlike the variables `E1,...,Em`—are not existentially quantified within N. (The variables in U are quantified outside N and should not be further bound when executing N.) N is executed as follows:

1. bodies `B1,...,Bn` are deduced from `B`. This is done as though a new rule-instance `h <- B` were executed to result in the following rule-instances:

   ```
   (h <- B1),...,(h <- Bn).
   ```

   (Here, `h` is a dummy, predicate symbol that does not appear elsewhere in the program.) This step guarantees that the existentially-quantified variables `E1,...,Em` are renamed differently in each `Bi`.

2. in each `Bi`, any further instantiations to variables that correspond to or are renamed versions of those in U are undone and rewritten as equalities or inequalities.

3. over each `Bi`, each variable renaming a variable X in U is (back-) replaced by X.

4. N is replaced by the following conjunction of literals:

   ```
   (not-exists E1',...,Em': (B1)),...,
   (not-exists E1'',...,Em'': (Bn))
   ```

This chapter has shown how active negation is carried out in Starlog. This active negation has been successfully implemented and tested in our Starlog interpreter, which is discussed in §7.1. In the following Chapter 6, we will contrast Starlog with related approaches.

Chapter 6

Comparison with Related Approaches

Our work has examined various aspects of Starlog: its formal semantics, explicit time and its use in expressing mutation and persistence, constructive negation and rewriting, and logical arithmetic. In this chapter, we relate a spectrum of approaches to the seemingly-disjoint parts of our work.

6.1 Partial Deduction

Partial evaluation, also called *projection*, is a transformation technique used to specialize a given program into a semantically equivalent and more efficient one based on partial, known input. The residual (resultant) program, when given the remaining input, behaves identically to the original when the latter is given the same, combined input. Partial evaluation is a general technique applicable to programs in any programming language, logic based or otherwise. According to Lam [Lam89, pages 5–6], partial evaluation originated in the 1950s and was first applied to logic programming in 1981. *Partial deduction* is partial evaluation as applied to programs in logic-based languages. It is mainly applied in the areas of meta-programming and compiler generation.

Partial deduction is intended as a static, preprocessing strategy. Typically, program rules are specialized to suit known, input goals using top-down, resolution-based *unfolding*, which substitutes a call with an instance of the definition. On the other hand, an X-computation of a Starlog program specializes one stage into the next. Starting with an initial stage, which is the set of all the program rules, an X-computation specializes during execution, i.e., dynamically. Starlog specializes by propagating partial, known bindings from heads of rule-instances into "goals" (body atoms) referring to them; thus, specialization information flows in a direction exactly opposite to that in unfolding.

Residual programs generated by partial execution of Starlog may be executed to completion, e.g., as Prolog programs. Also, if the pointers associated with atoms in the bodies of rule-instances are retained in such residual programs, redundant deduction can be reduced.

6.2 Prolog

6.2.1 Updates

Both Starlog and pure Prolog [CM81] are based on first-order, predicate calculus and share a model-theoretic semantics: a minimal model of rules in the program. Starlog and Prolog differ primarily in their approach to mutation and persistence of modelled objects: the problem of updates. This problem has to be tackled in applications, e.g., databases, that model persistent and mutable stores of data. Prolog uses `assert` and `retract` to force new rules into and remove rules from a program's database. Unfortunately, these navigational constructs do not have a consistent, operational meaning across different Prolog implementations, let alone a description within the static semantics of Prolog. Thus, they can be very confusing and error-prone in practice. Since Starlog explicitly incorporates time and the sequencing that it implies, it can directly model modifications to its database. Thus, Starlog includes the good from two worlds: it has a static semantics and can model change.

6.2.2 Negation

Negation in Starlog is not based on the passive, SLDNF inference rule [Llo87], as it is in Prolog. Instead, it is constructive, and aided by constrained arithmetic, it tries to force solutions. Unlike Prolog, Starlog retains tuples that have been computed and their deduction paths in the X-tree. This enables its `not-exists` predicate to be implemented in an efficient, semantically-clean fashion.

Prolog's "equivalent" of `not-exists` uses SLDNF negation and an all-solution retriever, such as `bagof`, `setof`, or `findall`. Due to standard Prolog's unfair search, these retrievers often execute inefficiently. They begin their search afresh each time and are executed to completion before they affect or are affected by other goals, i.e., they are not coroutining.

Memo-ization [Die87] is an optimization technique that computes the set of tuples satisfying a predicate and stores it in a table, known as the *extension table*, for later reuse. It has been introduced into some Prolog interpreters to improve termination and completeness in the presence of recursion. The ET* algorithm of [Die87] uses a least-fixpoint method to compute the "flat," extension table, but it only uses tuples for deduction. In contrast, Starlog's X-computation makes use of the "structured" X-tree, which carries the lineage information of tuples, and even non-tuple rule-instances to avoid recomputation in a more-general fashion.

6.2.3 Execution

Although their results are equivalent, SLD-resolution, which is used for executing pure-Prolog programs, differs greatly from Starlog's X-computation. (There are forms of Prolog-like languages, e.g., the Logic Data Language discussed ahead, that also use bottom-up, model-theoretic execution.) Perpetual processes, introduced earlier in §3.2.6, can be modelled in Starlog as a result of its concurrent search for deduction sequences; they would cause unbounded growth of standard Prolog's stack of choice points.

Unlike standard Prolog, where execution is "lazily" triggered by a query (goal), Starlog "eagerly" tries to compute a minimal model for the given program; a "goal" is deduced from this model. In Prolog, each answer is a separate binding of variables in a query rather than part of one overall, model-theoretic reply that Starlog aims to deduce.

Starlog does not explicitly backtrack over alternatives to search for solutions. However, it does provide a form of *or*-parallel search for alternative solutions.

6.3 Committed-Choice, Logic-Programming Languages

The committed-choice languages [Sha87, CG86b, Rin88] are a class of concurrent, logic-programming languages that attempt to model parallel, process-based execution. When executing a goal against a program written in these languages, if a choice of alternative clauses is encountered, then one of the alternative clauses is "arbitrarily" committed to for resolution. This "arbitrary" manner of choosing from among alternatives is called *don't-care* nondeterminism and is particularly suited to the efficient execution of perpetual processes, e.g., operating systems. Such perpetual processes cause an unbounded growth of standard Prolog's stack of choice points and need special handling. Here, we consider *Concurrent Prolog* [Sha87] as an example of a committed-choice language and compare it with Starlog.

Each clause in a Concurrent-Prolog program has in its body a sequence of atoms, called a *guard*, followed by the commit operator '|' and another sequence of atoms. Guards of clauses forming a predicate definition should be executed fairly, preferably in parallel for greater speed, when deciding which clause should be used for resolution. Fairness is required to enable at least one guard to succeed quickly even in the presence of guards that are complicated and that have non-terminating sub-refutations. The clause whose guard is satisfied "first" is selected. This parallel execution of guards is the *or*-parallelism exploited in Concurrent Prolog. Atoms in a conjunction may be executed concurrently to exploit *and*-parallelism.

Neither completeness nor soundness is lost in Starlog's exploitation of inherent parallelism since deduction paths are not arbitrarily discarded. The *don't-care* nature of Concurrent Prolog renders it incomplete—spurious failures can result even in the absence of negation. As a result, in the presence of negation, answers may be unsound.

The committed-choice languages offer elegant, programming techniques such as streams with partially-completed messages. *Streams* are pathways of communication handled through variable names, which are "anonymous" or relative names in that they can be bound to arbitrary terms. "Producers" and "consumers" are fixed for each stream via mode declarations; so, streams are directed pathways. Seen by the programmer as lists, streams allow the shipping of list elements from "producer" to "consumer" even before all the elements have been created. In contrast and like Linda [CG86a, CG89], Starlog must generate unique names for its global or public communication between rule-instances, which can be viewed as independently-executing processes. These names are full atoms including a predicate symbol, which is an absolute name.

Committed-choice languages depart significantly from the underlying, static semantics of Prolog. Their complicated semantics have caused non-trivial problems in implementing them correctly. For Concurrent Prolog, [Sar85] discusses the problems with *or*-fairness and the read-only, control annotation '?', which restrains the eagerness of unification. Although Starlog, as of now, is far from being a full-fledged, programming language like Concurrent Prolog, it has a strong, declarative basis.

6.4 Logic Data Language

Deductive-database systems employ logic to perform some of their functions, such as query handling and maintenance of data integrity. Integrity constraints of a database can be functional dependencies between domains, restrictions on domain values or typing, etc. Deductive databases reflect a confluence of database theory and logic programming and are of interest for many reasons, the most important being that they advocate a demarcation between declarative and procedural concepts, e.g., with respect to queries [Llo87].

Standard Prolog is navigational—the ordering of rules and goals is important for efficiency, termination, and correct operation of non-Horn constructs, such as updates. This is the major reason for failure to amalgamate Prolog with database languages in spite of the existence of declarative, database languages [Zan88], e.g., SQL, wherein the underlying, abstract machine determines an efficient control of the problem solver or query manager. With this in mind, *Logic Data Language* (LDL) [Zan88] was designed for deductive databases.

LDL retains the Horn-logic basis of Prolog while extending it with sets and update operators. Unlike Starlog, LDL distinguishes between predicates that refer to the program's database of *ground* facts and predicates that do not, and it uses matching instead of general unification. The schema for the facts' database and the facts themselves are managed by a separate, conventional, relational-database manager. These decisions were taken to make LDL efficient for data-intensive, secondary-storage-based applications. Further, LDL employs `magic-counting` methods [SZ87] for partial deduction based on partial, known inputs in queries. For recursive rules, LDL computes least fixpoints iteratively; this enables it to exploit the *or*-parallelism in database applications.

LDL's use of bottom-up, least-fixpoint computation makes it similar to Starlog. It uses relational operators, such as `join` and `project`, when combining answers from calls to database predicates. Thus, in the classification of [Con87], LDL provides a form of *and*-parallelism similar to Starlog's and akin to the Sync model of [LM86].

LDL executes all queries as transactions, i.e., atomically. It offers operators, viz., $+$ and $-$, for destructively writing into and reading from the facts' database, but these operators do not have a static semantics. For example, it is not clear what should happen when concurrent updates take place on an object. Also, LDL does not allow tailoring of the meaning of update. Starlog, by virtue of its notion of time, offers more suitable primitives for fine control over update.

LDL's safe handling of non-linear, recursive predicates in the presence of a cyclic, facts' database, its avoidance of redundant computation, rendering of multiple usages of predicates via the `choice` predicate, and query-optimization strategy make LDL efficient and declarative.

When extending conventional, relational databases to more-general, non-ground, and deductive databases, the proof-theoretic view is more powerful [Llo87]; but with the improvements being made in efficiency, model-theoretic views are becoming serious contendors for deductive databases. LDL takes a model-theoretic view, but of ground, deductive databases. Starlog also takes a model-theoretic view and deduces even with non-ground data.

6.5 Linda

Linda [CG86a, CG89] is a collection of conceptually-simple primitives for inter-process communication in a distributed network of processes. It models a globally- and transparently- accessible (virtual) memory, which is organized into tuples. (Linda has inspired Starlog's use of the word "tuple.") Processes do not communicate directly with each other, but place their messages into this global, tuple space to be picked up by other processes. This mechanism is much like a global mailbox. Data and the code that operates on the data are treated alike.

Both Linda and Starlog consider the computational universe as consisting of entities that are universally accessible. In Linda's case, these entities are ground, data tuples and active processes, which after execution turn into data tuples. In Starlog's case, the entities comprise atoms and rules, which are true over time intervals. Linda allows data tuples to be read—via the `read` operator—by many processes; Starlog allows a true instance of a predicate to be used for unification with atoms in the bodies of rule-instances. Linda also allows tuples to be explicitly deleted, via its `in` operator; Starlog's tuples cannot be explicitly deleted and may only be garbage-collected when no rule-instance will use them. Linda achieves object mutation by deleting and creating new, data tuples; Starlog simulates mutation by making use of timestamps to access the most-recent instances of predicates. Although Linda's tuples are not associated with a globally-known, virtual time, a timestamp could be forced onto the tuples, but as Linda's underlying, abstract machine is "unaware" of time, the effect would be incomparable with the use of time in Starlog.

Whereas Starlog is a general-purpose, programming language, Linda is a vehicle for distributed communication, which could be used, e.g., to implement a distributed, Starlog interpreter.

6.6 Connection-Graph Theorem Proving

Unrestricted use of resolution leads to redundancy in answers. To eliminate this redundancy, some theorem provers interpret programs by using connection graphs [Kow79, Bib83]. The links and nodes in connection graphs are akin to those of static, predicate-call graphs, but are dynamically modified during program execution: links and nodes may be created and, in some cases, deleted. The X-tree used by Starlog may be viewed as a connection graph. Here, we compare the connection-graph theorem proving (CGP) method of [Kow79] with Starlog's X-computation.

CGP is a method for proving theorems, which are presented as queries, and unlike other theorem provers, e.g., [Bib83], it attempts to find all answers to a given theorem (goal). During execution, CGP exploits its basis on queries by deleting clauses that cannot lead to a refutation. In contrast, Starlog has no notion of queries.

Unlike Starlog, CGP uses resolution. Despite this, both CGP and Starlog depend on fairness to accomplish their goals. CGP depends on fair selection of arcs, which link clauses to possible input-clauses for resolution, and Starlog depends on fair selection of rule-instances for execution. CGP can be more efficient because of its greater control over scheduling. By selecting arcs referring to facts, it can simulate a bottom-up execution, and by selecting arcs originating from the program's goal, it can simulate a top-down execution.

Even when deducing forward with a rule-instance that refers to non-tuple input-rules, Starlog propagates—as bindings—information from the heads of the input-rules into the rule-instance being executed. This, coupled with the maintenance of clausal lineage, i.e., parent-child relationships within the X-tree, enables the "early" satisfaction of negations and the avoidance of much recomputation. Neither strategy is used by CGP.

CGP deletes clauses containing unlinked atoms. This simplifies the connection graph, reduces the search space, and makes it easier to find a solution [Kow79, page 164]. In §7.4.2, we propose a similar form of garbage collection for our Starlog interpreter.

CGP detects and deletes tautologies by using pseudo links. Our Starlog interpreter does not do this at present, but in §7.4.3, we suggest ways of incorporating it. Also, CGP is applicable to "arbitrary," clausal programs, not just the programs with single-headed rules that Starlog caters to.

6.7 Temporal Logics

A temporal logic is a logical calculus capable of handling inferences involving time. According to [Gal87], this area has developed according to two major philosophies:

- first-order or "detenser" approach, wherein time is just another variable in a first-order theory, and anything that exists, exists timelessly.
- modal or "tenser" approach, wherein temporal elements are accorded logical status by reducing modal notions, such as possibility and necessity, to temporal ones.

These approaches may be viewed as rivals or as allies [Gal87].

Tempura [Hal87] is a logic-programming language. It is an imperative, parallel-programming language that is based on a logic, but uses neither unification nor deduction for executing its programs. It is based on a subset of Moszkowski's Interval Temporal Logic [Hal87], which, being a modal logic, extends classical logic with operators to express the modal notions of possibility and necessity. Tempura supports algorithmic constructs such as loops, hierarchical specification (via `proj`), and destructive assignment, but its greatest offerings are claimed to be the verifiability of its programs, ease of modelling change and persistence of objects, and "true" parallelism rather than arbitrary interleaving. According to Galton's classification [Gal87], Tempura would be a "tenser" approach and Starlog would be more like a "detenser," although time in Starlog is a real-valued, *logic* variable and may be quantified both existentially and universally to lend meaning to the temporal existence of objects.

Tempura programs are executed by transformation. Based on a current state, which presumably comprises values of variables and procedure invocations, a next state—corresponding to a next instant of Tempura time—is computed based on the static program; this is called *reduction*. Synchronization is achieved either implicitly—due to lock-step execution—or by means of shared variables.

Starlog shares with Tempura the conviction that for expressing mutation and persistence, a notion of time has to be built into the interpreter. Therein, the similarity ends. From the discussion in [Hal87], it appears that an exhaustive search through Tempura time—an implicit interval of discrete states—for "answers" is "expensive" because execution proceeds by stepping through time. Starlog's X-computation does not step through time, although timestamps could be used for scheduling rule-instances §7.4.3. It exploits time's explicit, real-valued nature for improving negation. Further, the explicit nature of Starlog time permits clearer programs and opens up the possibility of modelling physical (real) time directly.

Potential parallelism has to be explicitly specified in Tempura programs in order to be exploited. It also seems that Tempura does not avoid redundant computation. For illustrating Tempura execution, Hale [Hal87] considers a "simple" term $\Box(A = 1)$, which says that it is always the case that $(A = 1)$. This example raises efficiency-related questions that are unanswered in that paper, e.g., whether the invariant assignment is repeated in each reduction step through Tempura time.

6.8 Constructive Negation

In Chapter 5, we discussed the uses of constructive negation: to enable logic programming rather than theorem proving and to use negations to constrain search—and thereby, improve efficiency—rather than to just test conditions. Constructive negation encourages symmetry between (the execution of) negative and positive literals in a program. Here, we compare Starlog's constructive negation with the resolution-based one discussed in [Cha88, Cha89], addressed henceforth as CN.

Both CN and Starlog execute a negative literal incrementally by negating the (partial) solution of a corresponding, positive literal. Consider a negation `not(B)`, where `B` is the body. Then, `B` might have an infinite number of solutions when it is not grounded. Therefore, it is essential that the execution of `B` can be arbitrarily terminated. CN suggests using a depth bound on the sub-refutation used to execute `B`. Starlog's execution of `B` terminates since it rewrites—as explained in Chapter 5—the (partial) solutions got from a single execution of a rule-instance `h <- B` and since each individual execution of a rule-instance terminates.

Given a negation `not(B)`, where `B` is the body, CN first executes `B` to get a (partial) solution involving equalities. It then normalizes the solution by removing redundant variables and equations, along with irrelevant inequalities. This normalization greatly improves efficiency, and we would like to incorporate it into our Starlog interpreter.

When negating (partial) solutions, both CN and Starlog rely on rewrites. Unlike CN, Starlog uses rewrites, such as partitioning, that exploit the real-valued nature of arithmetic variables to force solutions. As a result of choosing the logical, interval arithmetic of [Cle87] for our Starlog interpreter, arithmetic constraints on a variable are directly reflected in its internal representation as an arithmetic interval. This has the following advantages:

- some constraints can be "internalized" into the involved variables and deleted, thereby, reducing the need to awaken or thaw them whenever variables are instantiated.
- Starlog's unification is more expressive since many constraints are transferred when arithmetic variables are unified.

Due to the use of resolution, much computation is replicated in CN, and no track is kept of pre-computed answers. Suppose there is an inequality `A =/= B`, where `A` and `B` are arbitrary terms. CN defines that this inequality is *valid* if `A` and `B` are not unifiable, and it is *satisfiable* if it is valid or if `A` and `B` are unifiable only after binding variables; the inequality is *primitive* if it is satisfiable, but non-valid, i.e., if `A` and `B` are unifiable only after binding variables. According to CN [Cha89, page 480], primitive inequalities are never selected from a goal. It appears then that a goal of the form `f(X,Y) =/= f(5,a)` would not be further executed by CN. Starlog first (forward) rewrites such a primitive inequality as `not(f(X,Y) = f(5,a))` and then executes it to give `not(X = 5, Y = a)`, at the end of the forward phase. This is then inverted into `notreal(X); X < 5; X > 5; (X = 5, Y =/= a)`, which is a simplified result capable of forcing solutions.

CN precludes recursive, predicate calls through negations, and thus, it avoids the issue of stratification in normal programs. Starlog programs, on the other hand, often rely on recursion through negation, and therefore, it is important for Starlog to allow all temporally-stratified programs. Its clever arithmetic and avoidance of redundant, deduction sequences serve it well for this purpose.

In this chapter, we have compared Starlog with related approaches. In the following Chapter 7, we present our conclusions and suggest future work.

Chapter 7

Conclusions and Future Work

This far, we have taken a look at the Starlog language, informally traced through its execution, provided a procedural semantics to its programs, examined its active negation, and compared it with other approaches. It is time to wind up. This chapter first offers a glimpse into how we went about implementing our Starlog interpreter. There is always a future and work to be done. Accordingly, this chapter next suggests avenues for further research: enhancing programming ease, declarativeness, formal semantics, and performance. Finally, it presents our conclusions.

7.1 How We Implemented Starlog

7.1.1 Origins

A Prolog interpreter that supported delayed evaluation and propagation of goals was available to us. This interpreter was written in Scheme [RC86, Dyb87] by John Cleary, Alan Dewar, and Susan Rempel of the Department of Computer Science, University of Calgary. In this interpreter, by using `when` declarations, in the style of NU-Prolog [TZ88], a goal may be made to delay on a limited set of conditions involving variables that appear in the goal.

7.1.2 Building Starlog

First, we implemented and incorporated the logical, interval arithmetic of [Cle87] into the available, Prolog interpreter; a continuation-passing style of programming was used. Next, we implemented an interpreter for definite Starlog, which called the earlier interpreter for the backtracking execution of primitives implemented there. This interpreter shared the earlier one's parsing routines. Finally, we built in active negation to realize a Starlog interpreter and tested this interpreter with both contrived and realistic programs.

The original, Prolog interpreter took up a little over thousand lines of Scheme code. We took five months and a thousand lines of Scheme code to implement the arithmetic. It took an additional six months and about two-thousand lines in Scheme to produce the Starlog interpreter. The entire interpreter, therefore, took up a little over four-thousand lines. Its constrained arithmetic and active negation, especially partitioning, are its distinguishing features. Carrying out rewrites for negation and keeping track of variable renaming when executing the body of a negated literal were the more-difficult parts to program.

Our Starlog interpreter is intended as a prototype—to give assurances that Starlog is a practical, programming language. We have tried to keep its implementation simple and have neither evaluated nor optimized its memory consumption or speed. Building the interpreter has given good insight into Starlog execution and helped in verifying its formal correctness in Chapter 4.

7.1.3 Tuples in Practice

We have treated tuples as heads of unit rule-instances, but in practice, we have to allow non-unit rule-instances to be considered as tuples. This may be due to insufficient constraints in the body of the rule-instance or due to the inability of primitives and negation to force solutions as demanded by a declarative reading of the program. In our interpreter, for a rule-instance R to be classified as a tuple, R has to satisfy both the following conditions:

- there should be no user-defined predicate calls in R's body.
- no further instantiation of variables should result when R is executed. This ensures that each primitive in the body has been executed, and that no improvement is possible—under the circumstances—by further executing R.

So, it is possible, for example, for a rule-instance with a `not-exists` call in its body to be classified as a tuple. We avoided discussing non-unit tuples in this thesis in the hope of keeping the discussion simple.

Input-rules that are tuples are resolved against; non-tuple input-rules are treated as described before. When resolution is used, existentially-quantified variables may accumulate on execution of a negation's body. Rewrite F3 of Fig 5.2 is more meaningful in this context.

7.2 Declarativeness

Our Starlog interpreter does much to support programs as executable specifications, but many improvements are desired. In particular, negation and logical arithmetic need to be strengthened, i.e., made more active. This will improve the declarativeness of its programs.

7.2.1 Negation

Rewrite R9 of Fig 5.5, as formulated, may be used only at the top level of a rule-instance so that correctness is not sacrificed. Otherwise, to retain correctness, the existentially-quantified variables have to be propagated into the nesting, predicate call, and the more-complicated rewrites in Fig 7.1 have to be used. If the rewrites of

#	Argument Literal	Rewrite
E12	not(not-exists X,Y,...: (A, B, ...))	exists X,Y,...: (A, B, ...)
E13	not(exists X,Y,...: (A, B, ...))	not-exists X,Y,...: (A, B, ...)
E14	p(T,...) <- exists X,Y,...: (A, B, ...)	p(T,...) <- A, B, ...

Figure 7.1: Possible Inversion of Existential Negation.

Fig 7.1 are used, adequate, variable-renaming support has to be provided to move existential quantification outwards in a rule-instance.

The rewrite for partitioning, in Chapter 5, should be generalized to handle every universally-quantified, arithmetic variable in a rule-instance, or at least those that appear anywhere in the head. Other rewrites should also be investigated.

7.2.2 Logical Arithmetic

While the logical, interval arithmetic of [Cle87] is extremely clever, its use does affect real-world completeness of the Starlog interpreter. For example, the constraint `int(X)`, as formulated in [Cle87], is not active enough to instantiate `X` with integers over an interval of real values—if `X` lies in `[1.7,3.2)`, `int(X)` is not constructive enough to deduce that `X` is 2 or 3, but given that `Y` lies in `[1.7,2.2)`, `int(Y)` deduces that `Y` is 2.

Similar is the case with `X != Y`. Given that `X~(-inf,+inf) != Y~[2,3)`, '`!=`', as formulated in [Cle87], is unable to simplify into the following disjunction:

```
X~(-inf,2); X~[3,+inf); X~[2,3) != Y~[2,3)
```

This means that selection of the literal `not-exists Y~[2,3): (X != Y~[2,3))` results in a variant of the same literal—instead of the following:

```
not(X >= 2, 3 > X).
```

In this case, the "lack" in arithmetic constructiveness can be compensated for by a cleverer rewrite rule for `not-exists` literals. Since `!=` is not discerning enough when dealing with arithmetic intervals that are not point intervals, Rewrite R4, in both Fig 5.3 and Fig 5.4, prefers `(X > Y; Y > X)` over `(X != Y)`, although these rewrites are equivalent.

It appears that Cleary [Cle87] purposely suppresses the activity of arithmetic constraints in order to control combinatorial explosion, but he does provide a control primitive `split(X)`, which forces the search for solutions by iteratively splitting `X`'s interval into disjoint sub-intervals.

In general, the constructiveness of arithmetic and that of rewrite rules should complement one another. Ways of making the arithmetic more active, without sacrificing efficiency, should be investigated.

7.3 Formal Semantics

Chapter 4 gives a procedural semantics for definite (Starlog) programs. It proves that a fair clause-selection guarantees sound and complete execution of definite programs; the atom-selection need not be fair. We have not yet proved the correctness of Starlog negation. To include Starlog's active negation in this semantics, we need to consider fair selection of literals because finite failure of X-derivations becomes important in the presence of negation. Since Starlog rules need not be clauses, it is not generally possible to show that a conjunction of Starlog rules is unsatisfiable, i.e., has no model, by restricting attention to Herbrand interpretations [Llo87, pages 17, 39]. Therefore, more general interpretations need to be considered.

For a program P, [Llo87] defines the greatest fixpoint (gfp) of a function T_P. We feel that an analogue of X-computation can be constructed that attempts to compute tuples covering a program's "maximal" or gfp-model, i.e., a set of tuples that when grounded equal gfp(T_P). When computing gfp(T_P), the heads of all rules in P can be thought of as the initial "interpretation"—instead of the infinite, ground B_P used in [Llo87]. This would mean that the heads are taken to be true initially. The program can be applied on this set of heads to generate the next set of heads, and so on. In any stage in such a computation, the heads cover gfp(T_P). This computation can be useful for definite programs that either contain no function symbols or have the property that, for each clause C, each variable in C's body also appears in the head. Such definite programs have $T_P\downarrow\omega$ =gfp(T_P) [Llo87, page 67]. (lfp(T_P) need not equal gfp(T_P), even for definite programs.) We hope that this idea helps model perpetual processes and negation. It may also prove to be the lever needed for the formal semantics of negation.

7.4 Performance

7.4.1 Information Flow

As mentioned in §6.1, X-computations specialize an atom in the body of a rule-instance to the rule-instances it refers to. Being able to reverse the direction of information flow would help efficiency. It is worth investigating the resulting effect, i.e., of specializing rule-instances to the atoms that refer to them. The `magic-counting` methods of [SZ87] should be considered in this regard.

7.4.2 Garbage Collection

As an X-computation proceeds, we only need to maintain rule-instances that are referred to and that may be referred to at some later stage. This means that by working down from the *root* of the (partial) X-tree, each deduction path—X-derivation path, in the case of definite programs—that has either failed or succeeded may be considered for garbage collection, i.e., for reclaiming its memory. On a succeeded or failed path, a rule-instance and the links of its body atoms may be deleted if it and its ancestor rule-instances are not pointed at by any atom in the (partial) X-tree.

The list used for scheduling rule-instances in our interpreter is passive. A rule-instance may be executed even though the rule-instances pointed at by its body atoms have not been executed or have just yielded variants upon execution. We feel that an "inverted," Starlog-execution scheme wherein a rule-instance upon execution awakens rule-instances referring to it can avoid wasteful re-execution and ease the detection of loops.

7.4.3 Scheduling

Our interpreter schedules rule-instances on first-come-first-served (FCFS) basis. (Any fair-scheduling scheme will work correctly.) Rule-instances can also be scheduled in non-decreasing order of their earliest-satisfaction times, i.e., the lower bounds of their corresponding timestamp ranges or intervals; this makes it easier to detect loops and then eliminate "useless" rule-instances. The list of rule-instances to be scheduled then looks like an event-list in a discrete-event simulation. If a loop is detected within this list, i.e., the execution of the constituent rule-instances resulted only in variants, then action based on temporal-stratification requirements can be taken against the involved rule-instances and their X-derivation paths in the (partial) X-tree.

Our Starlog interpreter fails to terminate given the following "harmless," definite, Starlog program:

```
    p(T) <- T >= 0, p(T).
```

Failing to terminate not only lowers efficiency, but it also harms the satisfaction of negations. Scheduling using timestamps can improve termination, but care has to be taken to ensure that completeness is not sacrificed—even for a definite program, there might be infinite tuples to be generated at a particular time value!

7.4.4 Active Negation

Given a negation `not(B)`, where `B` is the body, CN [Cha88, Cha89] first executes `B` to get a (partial) solution involving equalities. It then normalizes the solution by removing redundant variables and equations, along with irrelevant inequalities. Its normalization technique is equally applicable to Starlog negation and should be considered in order to improve efficiency.

7.4.5 Parallel Execution

Starlog exploits inherent parallelism in logic programs. It exploits *or*-parallelism by concurrently exploring deduction sequences from different program rules; as mentioned in §3.2.6, it resembles the Sync model of [LM86] in its exploitation of *and*-parallelism. All this is with a sequential, Starlog interpreter. A distributed, Starlog interpreter can take advantage of this parallelism to speed up execution.

Nodes in an X-tree do not share variables. Nodes on non-overlapping paths need to communicate only for unification. During X-tree construction, only its leaves, which comprise a stage in an X-computation, need to be executed by the interpreter. Therefore, the interpreter may use timestamps beyond scheduling rule-instances and active negation. Based on ideas from Time Warp [Jef85], these timestamps can provide the basis for an optimistic, distributed, *or*-parallel execution of nodes in a distributed X-tree. Then, execution can smoothly shift from a sequential to a distributed or parallel environment.

7.5 Conclusions

This thesis has offered an insight into a model-theoretic execution of logic programs. It has formalized the execution of definite (Starlog) programs and proved the execution to be semantically correct. It has described a method for executing Starlog's active negation. By building a prototype, Starlog interpreter, we have demonstrated Starlog's feasibility, and in this thesis, we have pointed out directions that can be taken to shape Starlog into a practical, programming language.

Bibliography

[BF81] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 1. William Kaufmann, Inc., Stanford University, 1981.

[Bib83] Wolfgang Bibel. Matings in matrices. *Communications of the ACM*, 26(11):844–852, November 1983.

[CG86a] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[CG86b] Keith [L.] Clark and Steve Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Cha88] David Chan. Constructive negation based on the completed database. In Robert [A.] Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 1, pages 111–125. The MIT Press, 1988.

[Cha89] David Chan. An extension of constructive negation and its application in coroutining. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, volume 1, pages 477–493. The MIT Press, 1989.

[CK91] John G. Cleary and Vinit [N.] Kaushik. Updates in a temporal logic programming language. Research Report No. 91/427/11, The University of Calgary, Department of Computer Science, Calgary, Alberta, Canada T2N 1N4, March 1991. Submitted to ILPS-1991.

[Cle87] John G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, January 1987.

[Cle90] John G. Cleary. Colliding pucks solved using a temporal logic. In *Proceedings of the Conference on Distributed Simulation.* Society for Computer Simulation International, January 1990.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Con87] John S. Conery. *Parallel Execution of Logic Programs*, chapter 3, page 55. Kluwer Academic Publishers, 1987.

[Die87] Suzanne Wagner Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings: 1987 Symposium on Logic Programming*, pages 264–271. IEEE Computer Society, IEEE Computer Society Press, August 1987.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[Gal87] Antony Galton. Temporal logic and computer science: An overview. In Antony Galton, editor, *Temporal Logics and their Applications*, chapter 1, pages 1–52. Academic Press, 1987.

[Hal87] Roger Hale. Temporal logic programming. In Antony Galton, editor, *Temporal Logics and their Applications*, chapter 3, pages 91–119. Academic Press, 1987.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Kow79] Robert [A.] Kowalski. *Logic for Problem Solving*, chapter 8, pages 163–178. Artificial Intelligence Series 7. North-Holland, 1979.

[Kow85] Robert [A.] Kowalski. Directions for logic programming. In *Proceedings: 1985 Symposium on Logic Programming*, pages 2–7. IEEE Computer Society, IEEE Computer Society Press, July 1985.

[Lam89] John Kwong Kei Lam. Control structures in partial evaluation of pure Prolog. Master's thesis, University of Saskatchewan, Department of Computational Science, Saskatoon, Saskatchewan, Canada, September 1989.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, University of Melbourne, Australia, second, extended edition, 1987.

[LM86] Peyyun Peggy Li and Alain J. Martin. The Sync model: A parallel execution method for logic programming. In *Proceedings: 1986 Symposium on Logic Programming*, pages 223–234. IEEE Computer Society, IEEE Computer Society Press, September 1986.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[RC86] Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12), December 1986.

[Rin88] G. A. Ringwood. Parlog86 and the dining logicians. *Communications of the ACM*, 31(1):10–25, January 1988.

[Sar85] Vijay A. Saraswat. Problems with Concurrent Prolog. Technical Report CS-86-100, Carnegie-Mellon University, Department of Computing Science, USA, May 1985. This was revised January 1986.

[Sha87] Ehud [Y.] Shapiro. A subset of Concurrent Prolog and its interpreter. In *Concurrent Prolog: Collected Papers*, chapter 2, pages 27–83. The MIT Press, 1987.

[SZ87] Domenico Saccà and Carlo Zaniolo. Magic counting methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–59, 1987.

[TZ88] James A. Thom and Justin Zobel. NU-Prolog reference manual: Version 1.3. Technical Report 86/10, University of Melbourne, Machine Intelligence Project, Department of Computer Science, Australia, January 1988.

[WH81] Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp*. Addison-Wesley Publishing Company, 1981.

[Zan88] Carlo Zaniolo. Design and implementation of a logic based language for data intensive applications. In Robert [A.] Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 2, pages 1667–1687. The MIT Press, 1988.

Appendix A

Starlog Syntax

Following is a specification of Starlog's syntax in Backus-Naur Form (BNF). The symbols ::=, |, {, }, [, and] are meta-symbols belonging to the BNF formalism; they are not symbols of the Starlog language. The curly brackets, '{' and '}', denote possible repetition of the enclosed symbols zero or more times. The box brackets, '[' and ']', denote optional occurrence of the enclosed symbols. Terminal symbols are shown in bold face, e.g., **terminal**.

```
program          ::= { formula }
formula          ::= rule . | unit-rule .
rule             ::= head <- tail
unit-rule        ::= atom
head             ::= atom
tail             ::= open-conjunction
open-conjunction ::= exp { , exp }
exp              ::= literal | encl-conjunction | encl-disjunction | ( exp )
encl-conjunction ::= ( exp , exp { , exp } )
encl-disjunction ::= ( exp ; exp { ; exp } )
literal          ::= atom | not ( exp ) | not-exists var-list : ( exp )
var-list         ::= [ var { , var } ]
atom             ::= composite
composite        ::= functor | functor ( term { , term } )
term             ::= constant | var | composite
functor          ::= lower-id
constant         ::= integer | real | lower-id
var              ::= upper-id
upper-id         ::= upper-case-letter id
lower-id         ::= lower-case-letter id
id               ::= { letter | digit }
```

Appendix B

Example, Starlog Programs

Following are some of the interesting programs that our Starlog interpreter has executed.

B.1 Prime Numbers

% The following is a transparent, Starlog program for computing primes.
% It closely reflects the idea of deleting all multiples of each prime
% as it is generated. The constraint `J >= K` within the negation
% helps remove redundancy.

```
prime(I) <- integer(I), I >= 0,
            not-exists J,K: (prime(J), integer(K),
                             I > J, I > K, J >= K, *(J,K,I)).
integer(N) <- integer(Nprev), +(Nprev,1,N), N >= 2.
integer(2).
```

B.2 Bouncing Ball

% The following Starlog program simulates a bouncing ball. Its
% predicate `bounce(T,Vdown)` gives the downward velocity `Vdown`
% at time `T` of a bounce; predicate `traj(T,Height,Vel)`
% follows the ball's trajectory between bounces.
% The `traj` tuples computed by Starlog are non-unit rule-instances
% and have unsatisfied, arithmetic constraints in their bodies.
% The `bounce` tuples come out as unit rule-instances:
% `bounce(0,1)`, `bounce(2,0.5)`, `bounce(3,0.25)`,
% `bounce(3.5,0.125)`, ...

```
bounce(T,Vdown) <- T >= 0, Vdown >= 0, traj(T,0,Uup), 0 > Uup,
                   +(Uup,Udown,0), *(Vdown,2,Udown).
bounce(0,1).
traj(T,Height,Vel) <- T >= 0, Height >= 0, bounce(Tb,Udown),
                      Tb >= 0, T > Tb, Udown >= 0,
                      +(Tdiff,Tb,T), Tdiff > 0, +(Vel,Tdiff,Udown),
                      *(Z1,2,Tdiff), +(Z0,Z1,Udown),
                    *(Z0,Tdiff,Height).
```

B.3 Temporal-Database Update

% The following Starlog program maintains a simple, key-value, temporal
% database via a predicate db. Updates, given by update
% tuples, happen at various times and sometimes a request to increment
% the entire database comes along, via input tuples.
% This program can be executed by Starlog without using partitioning.
% db((1,2],a,4), db((2,3],a,5), and db((3,+inf),a,7)
% are the three db tuples computed by Starlog.
% update(1,a,4), update(3,a,7), and update(2,a,5)
% are the three update tuples computed by Starlog.

```
db(T,Key,Val) <- T >= 0, Ts >= 0, T > Ts,
                 update(Ts,Key,Val),
                 not-exists Tp,Any:  (Tp >= 0, T > Tp, Tp > Ts,
                                      update(Tp,Key,Any)).
input(2,incr).
update(1,a,4).
update(3,a,7).
update(T,Key,NewVal) <- T >= 0, db(T,Key,OldVal),
                        +(OldVal,1,NewVal), input(T,incr).
```