DEVELOPING AN INFERENCE ENGINE FOR ASET-PROLOG

by

MARY LYNN HEIDT, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science Department

THE UNIVERSITY OF TEXAS AT EL PASO

December 2001

Abstract

ASET-Prolog is a new knowledge representation language recently introduced by M. Gelfond[4]. ASET-Prolog is an extension of A-Prolog - the language of logic programs under the answer sets semantics[5]. In the last decade this language found multiple applications in various areas of AI and CS[1]. ASET-Prolog extends A-Prolog by sets and functions from sets to natural numbers. Even though ASET-Prolog does not add to the expressive power of A-Prolog it, in many cases, leads to simpler and more concise representation of knowledge. The language is similar to an extension of A-Prolog by choice and weight rules recently introduced by Niemela and Simons[14]. However, the two languages have different semantics. The precise relationship between the two is currently under investigation.

The goal of this work is to develop and implement an algorithm for computing answer sets of ASET-Prolog. We begin with a description of the language ASET-Prolog. We then introduce the algorithm *aset* used to compute answer sets of ASET-Prolog programs. Finally, we describe the inference engine, ASET-solver. This inference engine extends the corresponding inference engine for A-Prolog built at NMSU[12].

ASET-solver consists of two parts. The first part, *set_parse*, takes a program Π of ASET-Prolog as an input and produces its ground version, Π_g. It is a Prolog program which calls *lparse*, a program grounder for A-Prolog[16]. The second part, *aset*, computes the answer sets of Π_g, is written in C, and can be viewed as a substantial modification of the NMSU inference engine.

Table of Contents

List of Figures

Chapter 1

Introduction

Programming languages can be divided into two main categories. One of these categories is *algorithmic* in which the program calls upon the computer to perform a sequence of tasks. In contrast to this type of programming is *declarative* programming. The statements in a declarative program are based upon rules of logic and provide descriptions of objects of a domain and their properties. This set of statements is often called a *knowledge base* and the goal of declarative programming is to find models or consequences of this knowledge base. The work of computing these models or consequences is often done by an underlying inference engine. For example, Prolog is a logic programming language that has such an inference engine built into it. The programmer does not have to specify the steps of the computation and can therefore concentrate on the specification of the problem. It is this separation of logic from control that characterizes declarative programming [4, 6, 9]. Declarative programming languages need to meet certain requirements. Some of these requirements are:[4]

- The syntax should be simple and there should be a clear definition of the meaning of the program.
- Knowledge bases constructed in this language should be elaboration tolerant. This means that a small change in our knowledge of a domain should result in a small change to our formal knowledge base[8].

- Inference engines associated with declarative languages should be sufficiently general and efficient. It is often necessary to find a balance between the expressiveness of the language and the desired efficiency.

One such language is A-Prolog, a logic programming language under the answer set semantics[5]. It is a declarative programming language with syntax and semantics similar to Prolog and has the ability to represent a wide variety of problems, such as reasoning with incomplete knowledge and the causal effects of actions [2]. There are currently several inference engines for computing answer sets of A-Prolog programs [3, 15]. The efficiency of these engines has led to some important applications including the use of one inference engine, *Smodels*, in the development of a decision support system for the space shuttle [1].

ASET-Prolog is an extension of A-Prolog that adds to the language sets of terms and functions from these terms to natural numbers. The addition of sets simplifies both representation and reasoning. The system *Smodels* has extended the language of A-Prolog with choice and weight rules [11, 14]. This extension has a semantics similar to that of ASET-Prolog. However, we believe the representation in ASET-Prolog is both simpler and more general.

The contribution of this thesis is the development of an inference engine, *ASET-solver*, for ASET-Prolog. This development includes:

1. Construction of the algorithm to compute answer sets of programs in ASET-Prolog. The algorithm extends the basic stable models algorithm from [15].

2. Implementation of this algorithm.

The algorithm consists of two parts: *grounding*, *set-parse*, and *model-finder*, *aset*. The grounding algorithm, replacing variables in the program by object constants, is written in Prolog. It takes a program Π of ASET-Prolog Prolog as an input, turns it into a program of A-Prolog by removing all occurrences of sets, grounds the resulting program by *lparse* [16] (the grounding algorithm of *Smodels*), and then reinserts the sets. The model-finder, *aset*, extends the model-finding part of *Smodels*. Our particular implementation of this algorithm modifies a system developed at NMSU [12].

This thesis is organized in the following manner. Chapter 2 presents the syntax and semantics of ASET-Prolog. Chapter 3 presents the operations and algorithms used both in the grounding of ASET-Prolog programs and in the computation of the answer sets. Chapter 4 is a description of the data structures used in the computation.

Chapter 2

Syntax and Semantics of ASET-Prolog

ASET-Prolog is an extension of A-Prolog and therefore we will begin this section with a brief introduction to A-Prolog under answer set semantics [5, 4]. We will then introduce the syntax and semantics of ASET-Prolog. Finally, we will describe the variant of ASET-Prolog that has actually been implemented.

2.1 A-Prolog

The syntax of A-Prolog is determined by a signature $\sigma = \langle C, P \rangle$ where C and P are collections of object constants and predicate symbols respectively. Each predicate symbol is associated with an arity – the number of parameters of the corresponding relation. As usual by terms we mean object constants and variables, atoms are expressions of the form $p(t_1, \ldots, t_n)$, where $t's$ are terms and p is a predicate symbol of arity n. For ease of representation the sequence $t_1, \ldots, t_n$ will be written as $\overline{t}$. A logic program Π in A-Prolog is a collection of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n. \tag{2.1}$$

where l_i's are atoms, l_0 is an atom or the symbol $\perp$, which represents falsity, and *not* is a logical connective called *negation as failure* or *default negation.* We will call the expression *not* l_i a not-atom and atoms and not-atoms will be referred to as literals. The rule states that if a reasoner believes in $l_1, \ldots, l_m$ and has no reason to believe in $l_{m+1}, \ldots, l_n$ then he must believe in l_0. Literals $p(\bar{t})$ and *not* $p(\bar{t})$ are called contrary. A literal contrary to l is denoted by $\bar{l}$. Terms, literals, and rules not containing variables are called *ground*. Ground terms will be represented by lower case letters and variables will be represented by upper case letters [4]. If X is a set of literals from Π, we will say the literal l is *true* in X if $l \in X$ and l is false in X if $\bar{l} \notin X$. Otherwise l is undefined in X.

The answer set semantics of a logic program Π assigns to Π a collection of *answer sets* – consistent sets of ground literals corresponding to beliefs which can be built by a rational reasoner on the basis of rules of Π. A consistent set of literals is a set that does not contain contrary literals. In the construction of these beliefs the reasoner is assumed to be guided by the following informal principles:

- He should satisfy the rules of Π, understood as constraints of the form: *If one believes in the body of a rule one must believe in its head.*
- He cannot believe in $\perp$.
- He should adhere to the *rationality principle* which says that *one shall not believe anything he is not forced to believe.*

A rule with $\perp$ as the head is called a constraint rule. Often a constraint rule is written omitting $\perp$.

The precise definition of answer sets will be first given for programs whose rules do not contain default negation. Let Π be such a program and let X be a set of ground literals from Π. We say that X is *closed* under Π if, for every rule *head* $\leftarrow$ *body* of Π, head is true in X whenever *body* is true in X. (For a constraint this condition means that the body is not contained in X.)

Definition 1 *(Answer set – part one)*

A consistent set of ground literals X of Π is an *answer set* for Π if X is a minimal set closed under Π.

It is clear that a program without default negation can have at most one answer set.

Example 1 *The program*

$$\Pi = \{p(a).\ \ q(X) \leftarrow p(X).\ \ \}$$

has the unique answer set $X = \{p(a), q(a)\}$.

To extend this definition to programs containing default negation, take any program Π, and let X be a set of ground literals from Π. The *reduct*, Π^X, of Π relative to X is the set of rules

$$l_0 \leftarrow l_1, \ldots, l_m.$$

for all rules as in equation (2.1) in Π such that $l_{m+1}, \ldots, l_n \notin X$. Thus Π^X is a program without default negation.

Definition 2 *(Answer set – part two)*

A set of ground literals X of Π is an answer set for Π if X is an answer set for Π^X.

(The above definition found in [4] differs slightly from the original definition in [5], which allowed the inconsistent answer set.)

The following examples show answer sets for programs containing default negation [10].

Example 2 *The program*

$$\Pi_1 = \{p \leftarrow not\ q, r.\ \ r \leftarrow not\ s.\}$$

has an answer set $X = \{r, p\}$ *because*

$$\Pi^X = \{p \leftarrow r.\ \ r \leftarrow\}$$

and X is the unique answer set of Π^X. The program

$$\Pi_2 = \{p(a) \leftarrow not\ p(b).\ \ p(b) \leftarrow not\ p(a).\}$$

has two answer sets,$\{p(a)\}$ and $\{p(b)\}$. The programs

$$\Pi_3 = \{p(a) \leftarrow not\ p(a)\} and\ \Pi_4 = \{p(a).\quad \leftarrow p(a).\}$$

have no answer sets.

2.2 The Language of ASET-Prolog

ASET-Prolog is an extension of A-Prolog that simplifies representation and reasoning by the addition of sets and functions from sets to natural numbers. It is very close in semantics to A-Prolog with choice rules of [14]. Literals of A-Prolog of the form $p(\overline{X})$ will be called regular atoms or *r-atoms*. The language will be expanded by two new types of atoms:

- An *s-atom* is an expression of the form

$$\{\overline{X} : p(\overline{X}, \overline{Y})\} \subseteq \{\overline{X} : q(\overline{X}, \overline{Z})\} \tag{2.2}$$

  where $\overline{X}$ is a list of *bound* variables and $\overline{Y}$ and $\overline{Z}$ are lists of *free* variables. The set of variables $\{\overline{X}\}$ are disjoint from the sets $\{\overline{Y}\}$ and $\{\overline{Z}\}$.

- An *f-atom* is an expression of the form

$$is(f, \{\overline{X} : p(\overline{X}, \overline{Y})\}, t) \tag{2.3}$$

  with the same understanding of free and bound variables as in the s-atom. F-atoms are used to represent functions. The value of the function f for the corresponding set is represented by the integer t.

A set of *library functions* will be added to the language of A-Prolog. These are computable functions over a condition, $p(\overline{X})$, and a set of ground terms, S, in the language. An example of such a function is $card(p(\overline{X}), S)$ which computes the number of ground instances, $p(\overline{x})$, which are elements of S. This function is defined as

$$card(p(\overline{X}), S) =_{def} |\{p(\overline{t}) : p(\overline{t}) \in S\}| \tag{2.4}$$

Another function could be $sum(p(\overline{X}), S)$, which computes the sum of the ground instances, x, where x ranges over the set of integers, such that $p(x)$ is in S. This function would be defined as

$$sum(p(x), S) =_{def} \sum_{p(n) \in S} n \tag{2.5}$$

Programs in ASET-Prolog will be a collection of rules of the form (2.1), where $l_1, \ldots, l_n$ may be any atom in ASET-Prolog and l_0 must be either an r-atom or s-atom.

A *ground* s-atom or f-atom is an atom in which all free variables have been replaced by ground terms. If an s-atom as in 2.2 is the head of a rule in program Π, all ground instances of $p(\overline{X}, \overline{y})$ are said to be *s-defined* in Π. An *instance pair* of an ground s-atom is ground instances of $p(\overline{X}, \overline{y})$ and $q(\overline{X}, \overline{z})$ in which the corresponding bound variables have been replaced by identical ground terms. For example a ground atom sa of 2.2 could be

$$\{\overline{X} : p(\overline{X}, a)\} \subseteq \{\overline{X} : q(\overline{X}, b)\}$$

and an instance pair of sa

$$p(c, a) \text{ and } q(c, b).$$

We will say that $p(c, a)$ is the *companion* of $q(c, b)$ in sa. If sa is the head of a rule in Π, $p(c, a)$ would be *s-defined* in Π.

If X is a set of ground terms an ASET-Prolog program Π, a ground s-atom of the form $\{\overline{X} : p(\overline{X}, \overline{y})\} \subseteq \{\overline{X} : q(\overline{X}, \overline{z})\}$ is said to be *true* in X if for every instance pair belonging to this atom

$$p(\overline{x}, \overline{y}) \notin S \text{ or } q(\overline{x}, \overline{z}) \in S$$

If an instance pair does not meet this condition it is said to *falsify* the s-atom.

An f-atom of the form in definition 2.3 is *true* in S if

$$f(\{p(\overline{x}) \ : \ p(\overline{x}) \in S\}) = t$$

We can now give a semantics of ASET-Prolog by generalizing the notion of a stable model of A-Prolog.

Definition 3 *(Stable models of ASET-Prolog)*
Let S be a collection of ground r-atoms. By $se(\Pi, S)$ (read as "the set elimination of Π with respect to S") we mean the program obtained from Π by:

1. removing from Π all the rules whose bodies contain s-atoms or f-atoms not satisfied by S;

2. removing all remaining s-atoms and f-atoms from the bodies of the rules;

3. replacing rules of the form $l \leftarrow \Gamma$ where l is an s-atom not satisfied by S by rules $\leftarrow \Gamma$;

4. Replacing the remaining rules of the form: $\{\overline{x} \ : \ p(\overline{x})\} \subseteq \{\overline{x} \ : \ q(\overline{x})\} \leftarrow \Gamma$ by the rules $p(\overline{t}) \leftarrow \Gamma$ for each $p(\overline{t})$ from S.

We say that S is a stable model of Π if it is a stable model of $se(\Pi, S)$.

We will now give several examples of the use of ASET-Prolog.

Example 3 *(Computing the cardinality of sets)*
We are given a complete list of statements of the form located_in(C, S) (read as "a city C is located in a state S"),e.g.

$$located\_in(austin, tx).$$
$$located\_in(lubbock, tx).$$
$$located\_in(sacramento, ca).$$

We would like to define a relation, $num(N, S)$, which holds if N is the number of cities located in state S. Assuming that our library contains a function card *defined in equation 2.4, we use the following rule:*

$$num(N, S) \leftarrow is(card, \{X : located\_in(X, S)\}, N).$$

After the grounding, this rule will turn into the rules

$$num(i, tx) \leftarrow is(card, \{X : located\_in(X, tx)\}, i).$$
$$num(i, ca) \leftarrow is(card, \{X : located\_in(X, ca)\}, i).$$

where i's are integers from 0 to some maximum integer m. (Notice, that a variable X is bounded and hence not replaced by ground terms.) It is easy to check the program has exactly one stable model containing the above facts and atoms $num(2, tx)$ and $num(1, ca)$.

The next three examples are taken from [11]. They demonstrate the use of rules of the form:

$$\{\overline{x} \ : \ p(\overline{x})\} \subseteq \{\overline{x} \ : \ q(\overline{x}\} \leftarrow \Gamma \tag{2.6}$$

with s-atoms in the heads. Rules of this form are called *selection* rules and are read as follows: " If Γ holds in a set S of beliefs of an agent then any subset of the set $\{\overline{t} \ : \ q(\overline{t}) \in S\}$ may be the extent of $p(\overline{x})$ in S"[1] [4]. The next example demonstrates the use of selection rules:

[1]By the extent of $p(\overline{x})$ in S we mean the set of ground terms such that $p(\overline{t}) \in S$.

Example 4 *(Cliques)*

Suppose we have a graph defined by the set of facts of the form $node(X)$ *and* $edge(X, Y)$

$$node(a).$$
$$node(b).$$
$$node(c).$$
$$edge(a, b)$$

We would like to define a relation $clique(X)$*, i.e. to write a program* Π *of ASET-Prolog such that for any graph* G *represented as above, the set of nodes* N *is a clique of* G *iff there is a stable model* S *of* $\Pi \cup G$ *such that an atom* $clique(t) \in S$ *iff* $t \in N$*. Recall that a set of nodes of a graph* G *is called a clique if* every two nodes from this set are connected by an edge of G. *This can be easily expressed by the following rules:*

$$\{X \ : \ clique(X)\} \subseteq \{X \ : \ node(X)\}.$$
$$\leftarrow clique(X), clique(Y), X \neq Y, not\ edge(X, Y).$$

Answer sets of the program consisting of graph G *combined with the first rule correspond to arbitrary subsets of nodes of* G*. Adding the constraint eliminates those which do not form a clique.*

The next example demonstrates how selection rules combined with cardinality constraints can allow selection of subsets of given cardinality.

Example 5 *(Coloring the graphs)*

Suppose we have a graph G *defined by the set of facts of the form* $node(X)$ *and* $edge(X, Y)$ *as in example (4) together with a set* C *of colors*

$$color(red). \quad color(green). \quad \ldots$$

We would like to color the graph in a way which guarantees that no two neighboring nodes have the same color. To this end we introduce a program Π *defining a relation* *colored(Node, Color) such that every coloring will be represented by the atoms of the form colored(n, c) from some stable model of* $\Pi \cup G \cup C$. *Program* Π *will consist of the following rules:*

$$
\begin{array}{rcl}
\{C : colored(X, C)\} \subseteq \{C : color(C)\} & \leftarrow & node(X). \\
& \leftarrow & is(card, \{C \ : \ colored(X, C)\}, N), \\
& & N \neq 1. \\
& \leftarrow & colored(X, C), \\
& & colored(Y, C), \\
& & edge(X, Y).
\end{array}
$$

The first rule allows the selection of arbitrary sets of colors for a given node X. *The second limits the selection to one color per node. The third eliminates the selections which color neighbors by the same color. The selections left after this pruning correspond to acceptable colorings.*

Finally, we will demonstrate how to use our language to express resource or cost constraints. This problem led the authors of [11] to the introduction of the *weight constraint*. The following example illustrates the use of such constraints.

Example 6 *(Knapsack problem)*
Suppose we are given a finite universe U *of objects, two relations* $weight(O, W)$ *and* $cost(O, C)$ *defining the weight and the cost of an object* O, *two numbers,* w *and* c, *and some condition* $p(X)$. *We want to define collections of elements of* U *which satisfy* $p(X)$, *have sums of their weights less than* w *and sums of their costs greater than* c. *Every such collection will be given by atoms of the form* $selected(o)$ *from some stable model of a program* Π *containing basic facts above and the rules:*

$$\{X : selected(X)\} \subseteq \{X : p(X)\}.$$

$$\begin{aligned} t1(W) &\leftarrow selected(X), \\ &\quad\;\; weight(X, W). \\ &\leftarrow is(sum, \{W : t1(W)\}, N), \\ &\quad\;\; N \geq w. \\ t2(C) &\leftarrow selected(X), \\ &\quad\;\; cost(X, C). \\ &\leftarrow is(sum, \{C : t2(C)\}, M), \\ &\quad\;\; M \leq c. \end{aligned}$$

The final example illustrates the use of s-atoms in the body of rules.

Example 7 *(Checking the course prerequisites)*

Suppose that we have a record of courses passed by a student, s, given by a collection of atoms

$$passed(s, c1). \quad passed(s, c2). \quad passed(s, c3).$$

and a list of prerequisites for each class

$$prereq(c1, c4). \quad prereq(c2, c4). \quad prereq(c4, c5).$$

Our goal is to express the following rule: A student S is allowed to take class C if he passed all the prerequisites for C and didn't pass C yet. *This rule can be written as*

$$\begin{aligned} can\_take(S, C) \;\leftarrow\; & \{X : prereq(X, C)\} \subseteq \{X : passed(S, X)\}, \\ & not\ passed(S, C). \end{aligned}$$

It is easy to check that the stable model M of this program, Π, consists of the above facts and an atom $can\_take(s, c4)$. Indeed, after grounding the above rule will turn into rules:

$$can\_take(s, c_i) \leftarrow \{X : prereq(X, c_i)\} \subseteq \{X : passed(s, X)\},$$
$$not\ passed(s, c_i).$$

where $0 \leq i \leq 5$. *The s-atoms in the bodies of the rules are satisfied for i = 1,2,3, and 4 and are not satisfied for i=5. So* $se(\Pi, M)$ *consists of the facts and rules*

$$can\_take(s, c_i) \leftarrow not\ passed(s, c_i).$$

where i = 1..4. It is easy to check that M *is the only stable model of this program.*

2.3 The Implemented Language

The description of the implementation that follows is for a variant of ASET-Prolog which will be called ASET-Prolog$^+$. The language of ASET-Prolog$^+$ is the language of ASET-Prolog with three modifications:

- The s-atoms and f-atoms will be allowed to have only one bound variable. Therefore, s-atoms will be of the form

  $$\{X : p(X, \overline{Y})\} \subseteq \{X : q(X, \overline{Z})\}$$

  and f-atoms will be of the form

  $$is(f, \{X : p(X, \overline{Y})\}, t).$$

- F-atoms may only appear as positive literals in the bodies of rules and may only represent monotonically increasing or decreasing functions. The restriction on monotonicity will allow for a more efficient implementation.

- The following declarations and statements from the language of *lparse* will be added to our language:

– **constant declarations** of the form

$$\text{const } num = expr.$$

where *expr.* is any constant value expression.

– **ranges** which are a shorthand method of defining numeric domains. The rule $num(1..3)$. is a shortcut for

$$num(1).$$
$$num(2).$$
$$xnum(3).$$

– **compute statements** of the form

$$\text{compute } N, \{L\}.$$

where N is an integer representing the number of answer sets to be computed with 0 indicating all answer sets, and L is a list of ground r-atoms or not r-atoms to be included in all answer sets. This statement is slightly different than the *lparse* compute statement, which contains no comma.

Chapter 3

Algorithms

Our goal is to develop a system, *ASET-solver* that will take a program written in ASET-Prolog$^+$ and calculate its answer sets. The system to compute answer sets for ASET-Prolog$^+$ programs will consist of two parts:

1. The grounding stage, *set_parse* which, through a series of transformations and the use of *lparse* grounds all free variables of a program of ASET-Prolog$^+$.

2. The computation stage, *aset* which will compute answer sets for the ground program.

This chapter will present the operations and algorithms that are used for each of these parts.

3.1 Grounding

We wish to make use of an already existing program, *lparse*, [17] to ground the free variables of an ASET-Prolog$^+$ program. Because *lparse* accepts programs of A-Prolog, transformations must be made to ASET-Prolog$^+$ programs. All sets must be removed while, at the same time, the meaning of these sets must be preserved. This process is accomplished through a Prolog program, *set_parse*. *Set_parse* takes as input an ASET-Prolog$^+$ program, Π, and produces as output a program, Π_g with all free variables grounded that is in the form needed for the computation stage. The grounding process by *set_parse* consists of five phases:

1. an ASET-Prolog$^+$ program Π_1 is read and stored in memory;

2. a first transformation, *transform1*, takes the program in memory and replaces all s-atoms and f-atoms with r-atoms, producing an A-Prolog program Π_2, and creates a collection R of records of these changes;

3. the program Π_2 is grounded by *lparse*;

4. a second transformation, *transform2*, takes $ground(\Pi_2)$ and using records of R reinserts the s-atoms and f-atoms with proper grounding of the free variables;

5. the program produced by *transform2* is translated into a form that can be used by the computational stage.

The first four steps will be described in the following sections and the final form of the output from *set_parse* will be described in chapter 4.

3.1.1 Reading Programs in ASET-Prolog$^+$

The initial phase of *set_parse* reads an ASET-Prolog$^+$ program, Π, a statement at a time and adds these statements to memory using the Prolog predicate assert. Statements asserted will be in an intermediate language $\mathcal{L}$. Statements of $\mathcal{L}$ may be of the form:

1. $rule(R)$, where R is a Prolog statement,

2. $lparse\_decl(LP)$ where LP is a statement or declaration from the language of *lparse*, or

3. $entry(X, Newatom, \text{ASET-Prolog}^+ atom)$, an entry recording changes to rules of Π.

These statements will be referred to as *rules*, *declarations*, and *entries* of $\mathcal{L}$. Rules and declarations of $\mathcal{L}$ will be asserted in this initial phase and entries of $\mathcal{L}$ will be asserted during *transform1* and are described in the following section.

3.1.2 Transform1

Transform1 will take a program Π in language $\mathcal{L}$ as input and produce as output a program Π' containing no sets and entries to memory recording the changes made to the rules of Π. Before describing the operations needed to accomplish this transformation, we will introduce the notion of an *element* of Π. We will use the following notation:

For a rule r of the form (2.1),

$head(r) = l_0$ and

$body(r) = \{l_1, \ldots, l_m, l_{m+1}, \ldots, l_n\}$.

Definition 4 *Let r be a rule of $\mathcal{L}$ in program Π and T be a term in the language of ASET-Prolog. By an element of Π we mean a pair (r, T) where*

(a) *T is an s-atom and $head(r) = T$, or*

(b) *T is an s-atom or f-atom and $T \in body(r)$, or*

(c) *T is an atom s-defined by Π and $T \in body(r)$.*

These three types of elements will be referred to as head-element*,* body-element*, and* s-defined-element *respectively.*

Example 8 *Consider a program Π consisting of rules $r1$ and $r2$ below:*

$$\begin{array}{rl} \{X \ : \ p(X,a)\} \subseteq \{X \ : \ q(X)\} & :- \quad s(a,b) \\ & :- \quad p(c,a), r(a,c). \end{array}$$

It is easy to see that a pair $(r1, \{X : p(X,a)\} \subseteq \{X : q(X)\})$ *is a head-element of* Π *and* $(r2, p(c,a))$ *is an s-defined-element of* Π.

We can now define the operation $\alpha_1(\Pi)$ that is used to construct the output of *transform1*.

Definition 5 *Let* Π *be a program in language* $\mathcal{L}$. *If* Π *contains no elements then*

$$\alpha_1(\Pi) = \Pi$$

otherwise, let $\Pi = \Pi_0 \cup r$, *where* (r,T) *is an element of* Π, *then*

$$\alpha_1(\Pi) = \alpha_1(\Pi_0 + \beta_1(r,T))$$

The operation $\beta_1(r,T)$, defined below, actually performs the work of the transformation process and will be called by $\alpha_1(\Pi)$ until Π contains no elements. The replacement process performed by $\beta_1(r,T)$ will make use of two operations, name(e) and newrule(r,T).

Definition 6 *Let* $e = (r,T)$ *be an element of* Π, n *be a function symbol not occurring in* Π, *and* $\overline{X}$ *represent the sequence of free variables in* T. *Then*

$$name(e) = n(\overline{X})$$

Definition 7 *Let* r *be a rule of ASET-Prolog and* $l \in body(r)$ *be an r-atom. Then*

$$newrule(r,l) = r'$$

where $head(r') = l$ *and* $body(r') = body(r)$.

Now we can define a replacement step, $\beta_1(r,T)$.

Definition 8 *Let $e = (r, T)$ be an element of* Π

$$\beta_1(e) = \begin{cases} \begin{cases} r' \left| \begin{array}{l} T \\ name(e) \end{array} \right. \\ entry(a, name(e), T). \end{cases} & \textit{if e is a head-element} \\ \\ \begin{cases} r' \left| \begin{array}{l} T \\ name(e) \end{array} \right. \\ rule(newrule(r', name(e))). \\ entry(b, name(e), T). \end{cases} & \textit{if e is a body-element} \\ \\ \begin{cases} rule(r) \\ rule(newrule(r, l)). \end{cases} & \textit{if e is an s-defined-element} \end{cases}$$

Operation β_1, an element at a time, eliminates sets by replacing rule r with rule r'. Rule r' is obtained from r by replacing an s-atom or f-atom in r with an r-atom that has been generated by name(e). This replacement is represented as

$$r' \left| \begin{array}{l} T \\ name(e) \end{array} \right.$$

in definition 8. Additionally β_1 adds entries that record these replacements and will provide the information needed to properly ground these replaced atoms after *lparse* has grounded all free variables. The rules generated by newrule(r,T) are necessary for proper grounding by *lparse*. When an atom generated by name(e) replaces a body atom, this newly generated atom will not appear in the head of any rule in Π. This is interpreted by *lparse* as an atom that can never be true and the rule will be eliminated. The same is true for rules containing s-defined atoms in the body and is the reason for the addition of the extra rules prior to grounding.

The following example illustrates the operation β_1 on head-elements and s-defined-elements.

Example 9 *Suppose the following rules belong to a program* Π *in language* $\mathcal{L}$.

$$rule(\{X : p(X)\} \subseteq \{X : q(X, Y)\} :- r(Y)). \tag{3.1}$$

$$rule(t(X, Y) :- p(X), s(X, Z), r(Y)). \tag{3.2}$$

Rule 3.1 along with the s-atom that is the head of this rule form a head-element of Π. *When the operation* β_1 *is applied to this element and, if the constant symbol generated by name(e) is new1, rule 3.1 is changed to*

$$rule(new1(Y) :- r(Y)). \tag{3.3}$$

and the entry

$$entry(a, new1(Y), \{X : p(X)\} \subseteq \{X : q(X, Y)\}). \tag{3.4}$$

is asserted to Π. *Rule 3.2 along with the term p(X) form an s-defined element of* Π. *When the operation* β_1 *is applied to this element the rule*

$$rule(p(X) :- p(X), s(X, Z), r(Y)).$$

is added to Π. *This is the additional rule that is necessary so that* lparse *will ground rule 3.2.*

The transformation of a rule containing an s-atom in the body is illustrated in the next example.

Example 10 *Suppose the following rule belongs to a program* Π *in language* $\mathcal{L}$.

$$rule(t(Y, Z) :- \{X : p(X, Z)\} \subseteq \{X : q(Y, X)\}, r(Y, Z)). \tag{3.5}$$

This rule along with the s-atom in the body form a body-element. When β_1 is applied to this element, and if the constant symbol generated by name(e) is new2, rule 3.5 is changed to

$$rule(t(Y,Z) :- \ new2(Z,Y), r(Y,Z)). \tag{3.6}$$

and the following rule and entry are asserted to Π.

$$rule(new2(Z,Y) :- \ new2(Z,Y), r(Y,Z)) \tag{3.7}$$

$$entry(b, new2(Z,Y), \{X : p(X,Z)\} \subseteq \{X : q(Y,X)\}). \tag{3.8}$$

It may not at first be obvious that it is necessary to define the operation α_1 in the manner set forth in definition 5 instead of using the simpler definition

$$\alpha_1(\Pi) = \alpha_1(\Pi_0) + \beta_1(r,T) \tag{3.9}$$

A problem arises with this definition if there exist in Π two elements (r,T) and (r,T'). After the application of $\beta_1(r,T)$, it is still necessary to apply β_1 to the element containing T'. If α_1 is defined using equation 3.9 this would not occur since α_1 is only applied to Π_0. The following example illustrates this problem.

Example 11 *Let the following rule, r, belong to program* Π *where r(Y,Z) is s-defined by* Π *and let* $\Pi = \Pi_0 + r$.

$$\{X : p(X,Y)\} \subseteq \{X : q(X)\} :- \ r(Y,Z), s(Y), t(Z).$$

If α_1 is defined as in equation 3.9, then $\alpha_1(\Pi)$ is $\alpha_1(\Pi_0)$ plus the following rule and entry, the results of $\beta_1(r, \{X : p(X,Y)\} \subseteq \{X : q(X)\})$.

$$rule(new1(Y) :- \ r(Y,Z), s(Y), t(Z)) \tag{3.10}$$

$$entry(a, new1(Y), \{X : p(X,Y)\} \subseteq \{X : q(X)\})$$

Since rule 3.10 contains an s-defined term, r(Y,Z), it is necessary to apply β_1 *to the element containing this rule and term. This application will not occur using equation 3.9 as the definition of* α_1 *and the rule*

$$rule(r(Y,Z) :- r(Y,Z), s(Y), t(Z))$$

will not be added to Π_0*. If r(Y,Z) is not the head of any rule in* Π*,* lparse *will believe the body of rule 3.10 to be obviously false and this rule will be eliminated from the ground program.*

.

3.1.3 Grounding by *lparse*

In order to ground the result of *transform1*, *set_parse* takes each rule or declaration of $\mathcal{L}$ and writes the corresponding Prolog or *lparse* statement to a temporary file. These rules and statements of $\mathcal{L}$ are removed from memory using the Prolog predicate retract. *Set_parse* makes a system call to *lparse* and the result, a ground program, is sent to another temporary file. This file is then read and the statements asserted to memory as in the initial phase of *set_parse*. This ground program will be referred to as Π_g.

3.1.4 Transform2

Transform2 will take a program Π_g in language $\mathcal{L}$ and reinsert the atoms removed by *transform1* with proper grounding of the free variables and remove rules generated by $newrule(r,l)$. In order to describe this transformation we need to introduce some terminology. Let T be a term of A-Prolog and T_g be a grounding of T. We will now introduce the notion of an *entrant* of Π_g.

Definition 9 *By an entrant of* Π_g *we mean a pair* (r, N_g) *where* $r \in \Pi_g$ *and*

(a) *entry*$(a, N, \_) \in \alpha_1(\Pi)$ *and head*$(r) = N_g$*, or*

(b) *entry*$(b, N, \_) \in \alpha_1(\Pi)$*,* $N_g \in$ *body*(r)*, and head*$(r) \neq N_g$*, or*

(c) *entry*$(b, N, \_) \in \alpha_1(\Pi)$*, head*$(r) = N_g$*, and* $N_g \in$ *body*(r)*.*

These three types of entrants will be referred to as a-entrant, b-entrant, and c-entrant respectively.

We can now define the operation $\alpha_2(\Pi_g)$ that is used to construct the output of *transform2*.

Definition 10 *Let* Π_g *be a program in language* $\mathcal{L}$*. If* Π_g *contains no entrants then*

$$\alpha_2(\Pi_g) = \Pi_g$$

otherwise, let $\Pi_g = \Pi'_g \cup r$*, where* (r, N_g) *is an entrant of* Π_g*, then*

$$\alpha_2(\Pi_g) = \alpha_2(\Pi'_g + \beta_2(r, N_g))$$

Similar to *transform1*, the operation $\beta_2(r, N_g)$ actually performs the work of the second transformation process and is called by $\alpha_2(\Pi_g)$ until Π_g contains no entrants. The replacement process performed by $\beta_2(r, N_g)$ will make use of a process *ground*(e, E).

Definition 11 *Let* Π_g *be a program in language* L *and* $e = (r, N_g)$ *be an entrant and* $E =$ *entry*$(\_, N, T)$ *be an entry of* $\alpha_1(\Pi)$*. If* S *is a mapping of the variables of* N *into the ground terms of* N_g *then*

$$ground(e, E) = S(T).$$

Since the free variables of term T are the arguments of term N, the substitutions found in the mapping S can be used by ground(e,E) to ground all the free variables of T.

Example 12 *Suppose* Π_g *contains the following rule,* r*, and entry,* $E \in \alpha_1(\Pi)$.

$$rule(new1(a,b) :-\ r(a), s(b)).$$
$$entry(a, new1(Y,Z), \{X : p(X,Y)\} \subseteq \{X : q(X,Z)\})$$

We can see that $e = (r, new1(a,b))$ *is an a-entrant of* Π_g*. The mapping,* S*, of the variables of* N *to the ground terms of* N_g *is* $Y = a$ *and* $Z = b$*. Therefore, the result of ground(e,E) is*

$$S(\{X : p(X,Y)\} \subseteq \{X : q(X,Z)\}) = \{X : p(X,a)\} \subseteq \{X : q(X,b)\}$$

We can now define a replacement step, $\beta_2(r, T)$.

Definition 12 *Let* $e = (r, N_g)$ *be an entrant of* Π_g

$$\beta_2(e) = \begin{cases} r' \left| \begin{array}{l} N_g \\ ground(e,E) \end{array} \right. & if\ e\ is\ an\ a{-}entrant \\ \\ r' \left| \begin{array}{l} N_g \\ ground(e,E) \end{array} \right. & if\ e\ is\ a\ b{-}entrant \\ \\ \emptyset & if\ e\ is\ a\ c{-}entrant \end{cases}$$

Operation β_2(e), an entrant at a time, transforms program Π_g into a program that contains sets by replacing rule r with rule r'. Rule r' is obtained from r by replacing a ground instance of a term generated by name(e) with a ground instance, generated by ground(e,E), of the original s-atom or f-atom. This replacement is represented as

$$r' \left| \begin{array}{l} N_g \\ ground(e,E) \end{array} \right.$$

in definition 12. In addition, β_2 removes the new rules added by $\beta_1(e)$ where e is a b-entrant. Note that the rules added for s-defined-elements are not removed. Suppose

$\Pi = \Pi_0 + r$ and r is one of these rules. The answer sets of Π will be the same as the answer sets of Π_0. The following example illustrates the result of β_2 on an a-entrant.

Example 13 *Consider example 9 and suppose the following rule, r, is a ground instance of rule (3.3) in* Π_g.

$$rule(new1(a) :-\ r(a)) \tag{3.11}$$

If $E =$ *entry 3.4 and* $e = (r, \_new1(a))$, *we have a substitution,* $Y = a$, *that can be used by ground(e,E). We then have*

$$ground(e, E) = \{X : p(X)\} \subseteq \{X : q(X, a)\}$$

and rule 3.11 is replaced in Π_q *by*

$$rule(\{X : p(X)\} \subseteq \{X : q(X, a)\} :-\ r(a)).$$

The transformation of b-entrants and c-entrants is illustrated in the next example.

Example 14 *Consider example 10 and suppose the following rules, r1 and r2, respectively, are ground instances of rules 3.6 and 3.7 in* Π_g.

$$rule(t(a, b) :-\ new2(b, a), r(a, b)) \tag{3.12}$$

$$rule(new2(b, a) :-\ new2(b, a), r(a, b)) \tag{3.13}$$

If $E =$ *entry 3.8 and* $e = (r1, new2(b, a))$ *we have the substitutions* $Y = a$ *and* $Z = b$ *that can be used by ground(e,E). We then have*

$$ground(e, E) = \{X : p(X, b)\} \subseteq \{X : q(a, X)\}$$

and rule 3.12 is replaced by

$$rule(t(a, b) :-\ \{X : p(X, b)\} \subseteq \{X : q(a, X)\}, r(a, b))$$

Since $e = (r2, new2(b, a))$ *is a c-entrant,* $\beta_2(e) = \emptyset$ *and* $\alpha_2(\Pi_g) = \alpha_2(\Pi_{g0})$. *The result of* β_2 *on this element is the removal for r2 from* Π_g.

The rules of Π_g now contain the rules of a ground program in ASET-Prolog$^+$.

3.2 Computation of Answer Sets

The algorithm for computating the answer sets of programs of ASET-Prolog is based upon the *smodels* algorithms for computing answer sets of A-Prolog.[15] The algorithm operates on a ground program, Π_g, of ASET-Prolog$^+$ and a set of literals B and computes answer sets of Π_g 'compatible' with B. (The precise definition of compatibilty will be given below.) If no such answer set exists the algorithm reports failure. Literals of B are extracted from the *compute* statement of Π_g. If we begin with an inconsistent B then Π_g has no answer sets.

Before we describe the algorithms used to compute answer sets, we need to introduce some terminology

- Given a set of literals A, A^+ will be used to refer to the set $\{a \mid a \in A\}$ and A^- will refer to the set $\{a \mid \text{not } a \in A\}$. We will define $Atoms(A) = A^+ \cup A^-$.
- A set of literals B *covers* a set of atoms A, $covers(B, A)$ if

$$A \subseteq Atoms(B)$$

- A set of r-atoms A *agrees* with a set of literals B if
  1. if $a \in A$ then $a \in B$, and
  2. if $a \notin A$ then $a \notin B$.
- A set A of r-atoms is *compatible* with a set of literals B if A agrees with B and s-literals and f-literals of B are true in A.

Definition 13 *Given a program* Π *and a set of literals* B*, the reduct of* Π *with respect to* B*,* Π_B *is*

$$\{h \leftarrow \; l_1 \ldots l_n \in \Pi \mid l_i \textit{ is not false in } B\}$$

Π_B is the set of all active rules of Π.

3.2.1 The Main Computation Cycle

The core algorithm, is executed by the function *aset* which, if $found = true$ returns an answer set that is compatible with B, otherwise the return value is undefined. This function forms the main loop of the computation process and is shown in figure 3.1. The functions *expand* and *pick* that are used in the algorithm will be described in more detail later. The algorithm:

1. pushes B on stack S of literals;
2. calls the function *expand* which adds additional atoms to S forming the set B' which satisfies the following properties:
   - B' is consistent,
   - $B \subseteq B'$ and
   - every answer set that is compatible with B is also compatible with B'.

   If no such B' exists the boolean variable $conflict$ is set to true and the stack S remains untouched. Otherwise, $conflict$ is set to false. The set B' is constructed by using the closure rules defined on the program and the set B. Complete description of these rules will be given in section 3.2.2.
3. If B' is inconsistent then there are no answer sets of Π that are compatible with B.
4. As long as S does not cover $Atoms(\Pi)$, then the loop containing the following steps are executed.
   (a) call the function pick which will choose a literal l undefined in S,
   (b) add l to S and call *expand*,

```
function aset(Π : program, B : set of literals, Found : boolean): set of r-atoms
{
    VAR S: stack of literals;
    initalize(S);
    push(B, S);
    expand(conflict, Π, S);
    if conflict then
        Found := FALSE;
        exit;
    while not covers(S, Atoms(Π)) do
        pick(l, S);
        push(l, S);
        expand(conflict, Π, S);
        if conflict then
            l := pop(S);
            push(not l, S);
            expand(conflict, Π, S);
            if conflict then
                Found := FALSE;
                exit;
    Found := TRUE;
    return S∩ r-atoms;
}
```

Figure 3.1: Main Cycle for Computing Answer Sets

(c) If expand discovers a conflict then no answer set compatible with B may contain l. Consequently, l is replaced by *not* l and *expand* is tried again.

(d) If a conflict is again discovered, then there is no answer set of Π which is compatible with B.

5. When the loop is exited, if $Found$ is true then the r-atoms of S form an answer set of Π containing B.

3.2.2 The *expand* Cycle

The *expand* function, illustrated in figure 3.2, operates on a set B of literals (normally located on the top of a stack S) and a program Π. It expands B to the set B' formed by repeating calls to functions *cl* and *obviously_false*. B' satisfies the following properties:

- B' is consistent,
- $B \subseteq B'$ and
- every answer set that is compatible with B is also compatible with B'.

Moreover, the program Π is updated by removing from it the rules falsified by literals from $B' \setminus B$ and by removing from its rules literals which belong to $B' \setminus B$. More precisely, *expand* returns the reduced program Π_B which is the reduct of Π with respect to B, as defined in definition 13. If there is no B' satisfying the above properties then the boolean variable $conflict$ is set to true and the set B remains untouched. Otherwise, $conflict$ is set to false.

Expand starts with a call to the function *cl* as illustrated in figure 3.3. This procedure returns the deductive closure of Π with respect to B.

```
function expand(conflict : boolean, Π :program, B : set of literals)
{
    VAR B_0, B' : set of literals;
    B_0 := B;
    repeat
        B' := B;
        B := cl(Π, B);
        B := B ∪ {not x | x ∈ Atoms(Π) and obviously_false(x)}
        conflict := conflict(B)
        if conflict then B := B_0;
        else Π := Π_B;
    until B' = B;
    return (Π, B);
}
```

Figure 3.2: The function expand

Definition 14 *A set U of atoms is called a* deductive closure *of B with respect to Π, ($cl(\Pi, B)$), if it is a a minimal set containing B and closed under the following seven rules:*[1]

1. *If all literals in the body of a rule $h \leftarrow l_1, \ldots, l_n$ are true in U, then*

$$h \in U$$

[1]These closure rules generalize the four original closure rules from the algorithm in [13].

procedure $\mathsf{cl}(\Pi$: Program, **VAR** B: stack of atoms)

{

Create(Q);

enqueue($closure(\Pi, B, Q)$);

while not empty(Q) **do**

dequeue(l, Q);

push(l, B);

enqueue($closure(\Pi, B, Q$);

}

Figure 3.3: The function cl

2. *If an r-atom l, is neither the head of any rule in Π, nor is s-defined in Π, then*

$$not\ l \in U$$

3. *If h is an r-atom that is the head of only one rule, $r = h \leftarrow l_1, \ldots, l_n$ in Π, $h \in U$, and h is not s-defined in Π, then*

$$l_1, \ldots, l_n \in U$$

4. *If h is the head of a rule $h \leftarrow l_1, \ldots, l_n$ in Π, not $h \in U$, all literals in the body except l_i belong to U, then*

$$not\ l_i \in U$$

5. *If $s = \{X : p(X, \overline{y})\} \subseteq \{X : q(X, \overline{z})\}$ is an s-atom*

   (a) *If $s \notin U$, and for any $t \neq t_i$* not *$p(t, \overline{y}) \in U$ or $q(t, \overline{z}) \in U$ then*

$$p(t_i, by) \in U \ and\ not\ q(t_i, \overline{z}) \in U$$

(b) *If* $s \in U$, *then for every* t

$$not\ p(t, \overline{y}) \in U\ \ or\ q(t, \overline{z}) \in U$$

6. *If* $s = \{X : p(X, \overline{y})\} \subseteq \{X : q(X, \overline{z})\}$ *is an s-atom and for every* t,not $p(t, \overline{y}) \in U$ *or* $q(t, \overline{z}) \in U$ *then*

$$s \in U$$

7. *If for every* t *from the domain of* $p(X)$, *either* $p(t) \in U$ *or not* $p(t) \in U$ *and the value of function* f *at* $\{t\ : p(t, \overline{y}) \in U\}$ *is* k *then*

   (a)

$$is(f, \{X : p(X, \overline{y})\}, k) \in U$$

   (b) *and for every* $m \in range(f)$ *such that* $m \neq k$

$$is(f, \{X : p(X, \overline{y})\}, m) \notin U$$

We say that a closure rule R is *ready to fire* with respect to B if B satisfies the *if* portion of R.

Notice, that rule 3 is not applied to program rules in which an s-atom is the head. This is illustrated in the following example.

Example 15 *Let* Π *be the program:*

$$\{X : p(X)\} \subseteq \{X : q(X)\} :-\ r.$$
$$q(a).$$

The only answer set of Π *according to definition 3 will be* $B = \{q(a)\}$. *If we add* r *to the answer set we will get two answer sets,* $B = \{q(a), r\}$ *and* $\{q(a), r, p(a)\}$.

procedure $\mathsf{closure}$(VAR Π: program, VAR B: set of literals

{

$R := \mathit{select\_rule}$; % selects closure rule ready to fire

if $R \neq \mathit{nil}$ **then**

$B := B \cup \mathit{head\_lit}(R)$;

$\Pi := \Pi_B$;

$\mathit{closure}(\Pi, B)$;

}

Figure 3.4: The closure procedure

Closure rule 7 can be extended to account for monotonic functions. This would lead to a more efficient implementation.

The procedure *closure*(Π, B) is illustrated in figure 3.4. It starts with selecting a closure rule R which is ready to fire with respect to B. If such an R is found then B is expanded by the result, $\mathit{head\_lit}(R)$, of fireing R, i.e. by the set of literals mentioned in the *then* portion of R. Finally, Π is replaced by the reduct, Π_B, of Π with respect to B and procedure *closure* is called again with the new parameters.

The second step *obviously_false* of *expand* is very close to the *smodels* at_most alogorithm. The goal of *obvioulsy_false* is to add to B', not a, for all atoms that are obviously false in B', using a function *compute_nant_dcl* that makes use of only the active rules of the program, Π_B. All s-atoms, f-atoms, and negated r-atoms are assumed to be true and a fixed point calculation determines which atoms can be derived. An atom can be derived if it is the head of or s-defined in an active rule in which all positive r-atoms can be derived. If an atom a cannot be derived from the current rules, not a is added to B'.

3.2.3 Detecting Conflicts

function *conflict*(B : set of literals) : boolean

{

if $B^+ \cap B^- \neq \emptyset$ or $\mathsf{conflict\_sets}(\Pi', B')$ **then**

return true;

else

return false;

}

Figure 3.5: Detecting Conflicts

The function *conflict*, illustrated in figure 3.5, ensures the consistency of the answer set by checking to see there is an atom a such that a and not a are present in the answer set. If B' is inconsistent then there is no answer set of Π' agreeing with B' and conflict returns false. We have added to conflict_sets a check on s-atoms and f-atoms in B'. Conflict_sets is true if there is a conflict between the value determined for an s-atom or f-atom from closure rules (1), (3), (4), and the evaluated value from rules (6), or (7). For instance if an s-atom, $\{X : p(X)\} \subseteq \{X : q(X)\}$ is the head of a rule where all the body literals are true in B, then this s-atom would be true in B due to rule(1). If $p(a)$and not $q(a) \in B$, then this s-atom would be evaluated false by rule(6) and conflict_sets would be true.

Finally, the algorithm for *pick* is illustrated in figure 3.6. The goal of *pick* is to select a literal to be added B. The selection of a literal will make use of two sets.

```
function pick(Π, B) : literal
% undefined if no literal is found
{
    VAR L: array of atoms
    L = Atoms(Π) \ Atoms(B);
    if (NAnt(Π) ∩ L ≠ ∅) then
        Select l ∈ L;
        return l;
    else if s-defined(Π) ∩ L ≠ ∅ then
        Select l ∈ L;
    else
        exit;
}
```

Figure 3.6: Function for selecting an arbitrary literal

- NAnt(Π): As in [15] this refers to the negative antecedents of program Π, the set of all atoms that appear negated in at least one rule of Π.

- s-defined(Π): the set of all atoms that are s-defined in Π.

We want to pick a literal, l, with the following properties:

- $l \notin B'$, and

- $l \in$ NAnt(Π) or $l \in$ s-defined(Π).

If no such a literal can be found there is no answer set compatible with B.

The following example illustrates the grounding of an ASET-prolog program and the computation of an answer set using the algorithm *aset*.

Example 16 *Consider program* Π *that specifies that one* X *should be selected, represented as* $selected(X)$*, that has the property* $p(X)$*. The statement, range(card,0 . . 3), specifies the range for the function.*

$$range(card, 0 \ .\ .\ 3).$$
$$\{X \ : \ selected(X)\} \ \subseteq \ \{X \ : \ p(X)\}.$$

$$\leftarrow \ is(card, \{X \ : \ selected(X)\}, N),$$
$$N \neq 1.$$

$$p(a). \qquad p(b). \qquad p(c).$$

The program set_parse *will change* Π *into its ground version,* Π_g*. Notice that there is no f-atom* $is(card, \{X \ : \ selected(X)\}, 1)$ *in the ground program, because this value is prohibited in the last rule of the original program.*

$$\{X \ : \ selected(X)\} \ \subseteq \ \{X \ : \ p(X)\}.$$

$$\leftarrow \ is(card, \{X \ : \ selected(X)\}, 0).$$
$$\leftarrow \ is(card, \{X \ : \ selected(X)\}, 2).$$
$$\leftarrow \ is(card, \{X \ : \ selected(X)\}, 3).$$

$$p(a). \qquad p(b). \qquad p(c).$$

$expand(conflict, \Pi_g, \emptyset)$ will compute B_1 such that:

$$\begin{aligned}
B_1^+ &= \{ \ p(a), p(b), p(c), \\
&\qquad \{X \ : \ selected(X)\} \subseteq \{X \ : \ p(X)\}\} \\
B_1^- &= \{ \ is(card, \{X \ : \ selected(X)\}, 0), \\
&\qquad is(card, \{X \ : \ selected(X)\}, 2), \\
&\qquad is(card, \{X \ : \ selected(X)\}, 3)\} \\
undecided(B_1) &= \{ \ is(card, \{X \ : \ selected(X)\}, 1), \\
&\qquad selected(a), selected(b), selected(c)\}
\end{aligned}$$

Suppose pick *returns not $selected(a)$. Them $expand(conflict, \Pi_g, S)$ adds nothing to S. If this process is repeated for not $selected(b)$ and not $selected(c)$ we will have:*

$$\begin{aligned}
B_2^+ &= B_1^+ \\
B_2^- &= B_1^- \cup \{selected(a), selected(b), selected(c)\} \\
undecided(B_2) &= \{is(card, \{X \ : \ selected(X)\}, 1)\}
\end{aligned}$$

Next call to expand *attempts to apply its inference rules resulting in*

$$\begin{aligned}
B_3^+ &= B_2^+ \cup is(card, \{X \ : \ selected(X)\}, 0)\} \\
B_3^- &= B_2^- \cup is(card, \{X \ : \ selected(X)\}, 1)\}
\end{aligned}$$

The set is inconsistent, and hence conflict *is set to TRUE and S is unchanged. The last literal guessed, not $selected(c)$, is removed from S and replaced by $selected(c)$. The final call to* expand *adds $\{is(card, \{X \ : \ selected(X)\}, 1)\}$ to S. Now all literals are decided and S contains an answer set of Π.*

Chapter 4

Implementation

The correct and efficient implementation of the algorithms described in chapeter 3 depends upon the choice of data structures. In this chapter we will describe the data structures used to implement the functions `expand`(to include the implementation of the closure rules and the detection of the `obviously_false` atoms), *push* and *pop*. The data structures used are similiar to those of [15, 12] but have been modified to accomodate sets.

4.1 Program Representation

The atoms and rules of a program are stored in arrays. The array *program_rules* contains the rules of the program. Atoms are stored in three separate arrays. The arrays *atoms, satoms* and *fatoms* contain the r-atoms, s-atoms, and f-atoms, respectively of the program. The atoms and rules are stored in the array so that the index of the array corresponds to number associated with the rule or atom stored in that position. r-atom i will be stored in *atoms[i]*.

In addition to the arrays there is a stack, *trail*, that holds the current answer set under construction. When an atom is determined to be true or false in the current answer set, a pointer to that atom is pushed on the stack. It is necessary to use a stack for the construction of the answer set because at times, during the construction of the answer set, a literal l is added that is produced by *pick* rather than *expand*.

When this atom is added to the answer set the value of the atom is guessed. If a conflict arises, it is necessary to remove all literals added after l. By using a stack these literals are on the top and will be the first removed.

The program uses three queues in the computation of answer sets.

posqueue is used by the *expand* operation to store atoms that have been determined to be true in the current answer set.

negqueue is used by the *expand* operation to store atoms that have been determined to be false in the current answer set.

closure_queue is used to determine the atoms that are obviously_false in the current answer set. It supports a final check used to verify that each atom present in the answer set is supported by the answer set.

The information associated with each rule that aids in the computation of the deductive closure is:

type The rule type: 1 if the head is an r-atom and 8 if the head is an s-atom.

negs and nbody Count of and array of pointers to negated r-atoms in the rule body.

poss and pbody Count of and array of pointers to positive r-atoms in the rule body.

neg_sas and nsbody Count of and array of pointers to negated s-atoms in the rule body.

sas and sbody Count of array and of pointers to positive s-atoms in the rule body.

fas and fbody Count and array of pointers to f-atoms in the rule body. Each entry in fbody contains the rule value of the f-atom.

head A pointer to the atom that is the head of the rule.

inactive Number of falsified body literals.

upper Number of positive body literals not known to be derivable.

lit Number of body literals not satisfied by the answer set under construction.

If we have the following rule

$$a \leftarrow b, c, \text{ not } d.$$

then we would have

$$\begin{array}{rclrclrcl} r.head & = & a & r.nbody & = & \{d\} & r.pbody & = & \{b, c\} \\ r.nsbody & = & \emptyset & r.sbody & = & \emptyset & r.fbody & = & \emptyset \end{array}$$

If the current answer set under construction is $B = \{b, d\}$ then

$$r.inactive = 1 \quad r.lit = 2$$

The counters *negs, poss, neg_sas, sas,* and *fas* begin as a count of the atoms in the arrays. The value of these counters will change once those atoms that will be contained in all answer sets are determined. These atoms in a program Π include:

- Atoms that will always be true in B, such as the facts of Π.
- Atoms that will always be false in B, such as atoms that are not the head of any rule in Π. These atoms will be added to B as not-atoms.
- Atoms and not-atoms added by means of a `compute` statement.

At this point all inactive rules are removed from the program and these counters are reset to the numbers of the atoms in the arrays that are undefined in B.

The additional information needed in *fas* for f-atoms is due to the manner of their grounding as illustrated in example 3. Remember that if we have a ground f-atom

fa, $is(f, \{X : p(X, \overline{y})\}, i)$, i represents a range of integers from 0 to some max and this atom will be grounded for all $i's$. We say i represents the *rule value* of fa in rule r. Therefore, when fa is fully evaluated it will be true in one rule where $i = val$ and false in all other rules.

The information associated with each r-atom a is:

name

isnant True if a is in NAnt(Π).

counthead and head Count of and array of pointers to rules in which a is head.

countpos and pos Count of and array of pointers to rules in which a is positive in the body.

countneg and neg Count of and array of pointers to rules in which a is negated in the body.

pairs A list containing information associated with any s-atoms for which a is an instance. The information included for each s-atom is:

- **sa** Pointer to an s-atom for which a is an instance.
- **left** True if a is an instance of the left set of sa and false if a is an instance of the right set.
- **companion** Pointer to the companion of a in sa.

computeTrue True if atom is required to be true in the compute statement.

computeFalse True if atom is required to be false in the compute statement.

headof Number of rules in head array where $rule.inactive = 0$.

sdefined Number of active rules in which a is s-defined.

Bpos If $Bpos$ = TRUE, $a \in B$

Bneg If $Bneg$ = TRUE, *not* $a \in B$

closure True if atom can be derived.

guessed True if atom added to trail as a result of *pick*.

f_pair A list containing information associated with any f-atoms for which a is an instance. The informaton included for each f-atom is:

- **fa** Pointer to an f-atom for which a is an instance.
- **bound** The value of the bound variable in a (0 if the bound variable is nonumeric)

The f-atoms and s-atoms contain similiar information to r-atoms . Both do not have the fields *name, pairs, fpairs, computeTrue, computeFalse, sdefined* and *closure.* Each s-atom sa contains the following additional fields:

countpairs and instances Count of and array containing the instance pairs of sa. The array contains the following information on each pair.

- **left** Pointer to the r-atom that is a ground instance of the left set of sa.
- **right** Pointer to the r-atom that is a ground instance of the right set of sa.

uncovered_pairs Number of unevaluated instance pairs.

false_pairs Number of instance pairs that falsify sa.

Each f-atom fa contains the following additional fields:

function number representing function identity.

pos This array contains additional information for f-atoms.

> **rules** A linked list of pointers to all rules containing the ground f-atom.
>
> **Bpos** True if ground f-atom true in B.
>
> **Bneg** False if ground f-atom false in B.

top Maximum index in pos.

instances The number of r-atoms that are ground instances of fa.

val The current value of the function.

notcovered The number of undecided instances.

The array *pos* for f-atoms will have indices corresponding to the range of the function. In our current implementation the i in an f-atom, $is(f, \{X : p(X, \overline{y})\}, i)$, represent integers ranging from 0 to 100 and $pos[t]$ will contain pointers to rules where $i = t$.

The arrays of atoms and rules are intialized from input provided by set_parse. This input is a numeric input with the exception of the names of the r-atoms. Information included in the input is:

List of Rules Each rule is written with a number representing each atom in the head and body. The type and value (atom or not-atom) of each atom is identified. f-atoms will also have attached an integer representing the value of the function in that rule.

List of r-atoms Each r-atom is represented by number and name.

List of s-atoms Each s-atom is represented by a number. In addition there is a list of all r-atoms that are ground instances of the s-atom.

List of f-atoms Each f-atom is represented by an atom number and a number representing the identity of the function. In addition there is a list of all r-atoms that are ground instances of the f-atom along with the value of the bound variable in each r-atom.

Compute Statement This is a list numbers of the atoms and not-atoms requested by the user to be in all computed answer sets. In addition there is an integer indicating the number of answer sets to be computed.

The code which reads the input can be found in the file *readin.c* and the entry point is the function *read_program*. The exact form of the input is contained in the comments above the functions that read the input.

The arrays are created as the input is read. As each rule is read the corresponding structure is created in *program_rules*, and a structure is created for each atom in the rule in one of the arrays *atoms, satoms*, or *fatoms*. When the list of r-atoms is read a name is associated with the atom. The fields *pairs* and *fpairs* in the r-atom structure and the field *instances* in the s-atom structure are initialized as the lists of s-atoms and f-atoms are read. At this time if an r-atom is a ground instance of any s-atom or f-atom pointers will be set from the r-atom to these atoms and from the s-atoms to the r-atom.

In the file *main.c* are functions that complete the initialization. The function *init_structures* completes the program representation by making sure each atom contains pointers to rules containing the atom and that each rule contains pointers to atoms of the rule. The function *init_system* creates the stack and three queues.

4.2 Computing Closure

Atoms are added to B using the seven closure rules described in section 3.2.2. This addition is recorded using the fields Bpos and Bneg and by placing the atom on the stack *trail*. For an atom a, if $a \in B$ then $a.Bpos$ is true, if $a \notin B$ then $a.Bneg$ is true, and if a is undecided in B both fields are false. This addition to B can occur during *expand* or as a result of *pick*. In this section we will describe how the atoms are selected for addition to B using the seven closure rules. This takes place in the function *expand*.

In the function *cl* atoms are removed from either *posqueue* or *negqueue*. There is first a check for conflit, and then the atom is checked to see if it has already been added to B. For example, when an atom a is removed from *posqueue* if $a.Bneg$ is true we have a conflict, and if $a.Bpos$ is true the atom has already been added to B. If there is no conflict and the atom is not already in B, the field $a.Bpos$ is set to true and a search begins for closure rules that are ready to fire. These rules are identified using the fields associated the atoms and rules. Any atoms that can be added as a result of firing these rules are placed in the queues. For a literal l the computation of the closure rules can be done in the following manner.

1. For every rule r in which l is an atom in the body decrement $r.lit$. If $r.lit = 0$, then $enqueue(r.head, posque)$.

2. For every rule r in which l is a not-atom in the body increment $r.inactive$. If $r.inactive = 1$ and if a is the head of r and is not the head of or s-defined in any other rule, then $enqueue(a, negqueue)$.

3. If $l \in B$, and l is an r-atom and the head of only one rule r ($l.headof = 1$, $l.sdefined = 0$), then for every atom a in the body of r, enqueue(a,posqueue) and for every not-atom a in the body of r enqueue(a,negqueue).

4. If $l \notin B$, and l is the head of only one rule r and if $r.lit = 1$, find the l_i in the body of r that is undefined in B. If l_i is an atom then $enqueue(l_i, negqueue)$, else $enqueue(l_i, posqueue)$.

5. If l is an s-atom

   (a) if $l \notin B$, $l.uncovered\_pairs = 1$, and $l.false\_pairs = 0$, then find an atom $l.instances[i]left$ or $l.instances[i]right$ that is undefined in B

      i. if $l.instances[i].right$ is undefined in B and $l.instances[i].left \in B$ then $enqueue(l.instances[i].right, negqueue)$

      ii. if $l.instances[i].left$ is undefined in$B$ and $l.instances[i].right \notin B$ then $enqueue(l.instances[i].right, posqueue)$.

   (b) If $l \in B$, for each instance pair , $0 \leq i < count\_pairs$

      i. If $l.instances[i].left \in B$, then $enqueue(l.instances[i].right, posqueue)$

      ii. If $l.instances[i].right \notin B$, then $enqueue(l.instances[i].left, negqueue)$

6. If l is an r-atom and in the list $l.pairs$ l has a companion defined in B

   (a) If this instance pair falsify an s-atom s then increment $s.false\_pairs$. If $s.false\_paris = 1$, then enqueue(s,negqueue).

   (b) else if $s.uncovered\_pairs = 0$ then $enqueue(sa, posqueue)$.

7. If l is an r-atom, for every f-atom f in the list $l.fpairs$, $t = f.val$ is calculated.

   (a) If $f.notcovered = 0$ then

      i. $enqueue(f.pos[t], posqueue)$

      ii. $enqueue(f.pos[i], negqueue)$ for all $i \neq t$.

4.3 Computing the Obviously False Atoms

Addition of atoms to B by *expand* is also done by computing the obviously false atoms in B. This computation makes use of the counter in each rule r, *r.upper*, which is the number of r-atoms in *r.pbody* not known to be derivable and the Boolean flag in each atom a, *a.closure*, which is true if a can be derived. Also used in the computation is the queue *closure_queue* and an array *tempfalse*.

Initially closure_queue is used to determine the r-atoms that will be false in all answer sets. This is determined as follows:

1. Facts of the program (heads of rules with no bodies) are in B. Those that are r-atoms become the initial atoms in *posqueue* and *closure_queue*. For those that are s-atoms, all r-atoms s-defined by this atom are enqueued in *closure_queue*.
2. In rules where the bodies contain only s-atoms, f-atoms, or negated r-atoms
   (a) If the head is an r-atom, then it is enqueued in *closure_queue*.
   (b) If the head is an s-atom, all r-atoms s-defined by the s-atom are enqueued in *closure_queue*.
3. For each atom a in *closure_queue*, if *a.closure* is set to false, then
   (a) *a.closure* is set to true.
   (b) For each rule r in *a.pos*
      i. Decrement *r.upper*.
      ii. If $r.upper = 0$, enqueue the head or s-defined atoms in *closure_queue*.
4. When *closure_queue* is empty, for each r-atom a with *a.closure* set to false, *not* a is added to B. These become the initial atoms in *negqueue*.

During the expand procedure, each time a rule becomes inactive its head or the s-defined atoms are enqueued in *closure_queue*. This indicates that the atom(s) possibly do not have a stable foundation. At the end of the expand cycle a function *compute_nant_dcl* again computes the atoms that must be false in B in the following manner:

1. For each atom a in *closure_queue* with $a.closure$ set to false:

   (a) For each rule r in $a.pos$ increment $r.upper$. If $r.upper = 1$, r is active and the head (or s-defined atoms) has positive closure, then enqueue the head (or s-defined atoms) in *closure_queue*.

   (b) Add a to *temp_false*.

2. The next step is to check if any atom a in *temp_false* can be derived, i.e. a must be in at least one rule r in which

   (a) a is the head of or s-defined in r.

   (b) $r.inactive = 0$.

   (c) $r.upper = 0$.

   All such atoms have *a.closure* set to true and the result propogated using *closure_queue*.

3. For each atom a in *temp_false* with $a.closure$ still set to false, if $a.Bneg$ is not set to true, $enqueue(a, negqueue)$.

4.4 Push and Pop

The *push* and *pop* procedures of the main computation cycle *aset* refer to addition of a literal produced by *pick* to *trail* and removal of this literal from *trail* if a conflict develops. When an literal is added to B and pushed on *trail*, the effects of this addition must be propogated. This is true if an atom is added as a result of *pick* or *expand*. Likewise, when an atom is removed from B, it is of course removed from *trail*, but also the effects of this removal are propogated. Therefore, *push* and *pop* refer not only to the actual addition to or removal from the stack, but also to the propogation of these additions and removals.

The propogation of *push* results in the firing of the closure rules as illustrated in the previous section. As in [15, 12] the main work for the computation of closure rules 1, 2, 3, and 4 is done by the functions *RuleMightFire(), RuleInactivate(), backchainTrue(),* and *backchainFalse()*. The functions *ForceAtomSatomFalse* and *ForceAtomsSatomTrue* are used for closure rule 5, *SetCheckFalseAtom* and *SetCheckTrueAtom* for rule 6 and *FatomCheckAtom* for rule 7. These procedures are called when an atom is added to B. A pseudocode is shown in figure 4.1 for the propogation of pushing an r-atom on trail. The propogation for for other types of atoms and not-atoms would be similar. When an literal a is pushed in *expand* the field *a.guessed* is set to false and when a is pushed as a result of *pick*, *a.guessed* is set to true, marking a choice point on trail.

When the *pop* procedure removes a literal from *trail* it must undo work of *push.* The functions *AtomBacktrack, SatomBacktrack* and *FatomBacktrack* actually perform perform this work. The pseudocode for *AtomBacktrack* is shown in figure 4.2. The function *AtomBacktrackSatom* that is called in this function will reset the counters *false_pairs* and *uncovered_pairs* and the function *AtomBacktrackFatom* will calculate the current value of functions following the removal an atom from B. The rule backtrack functions are responsible for resetting the rule counters *inactive, lit* and *upper.*

When a conflict is found during expand, atoms are popped from trail until an literal a is popped with the field *a.guessed* set to true. This literal is the most recently

procedure pushtrue(a: atom)

{

    for (all $r \in a.pos$)**do**

        RuleMightFire(r);

    for(all $r \in a.neg$)**do**

        RuleInactivate(r);

    if($a.headof = 1$ *and* $a.sdefined = 0$)**then**

        for the active rule r in a.head **do**

            backchainTrue(r);

    if($a.pairs \neq$ *null*) then

        for(all s-atoms $s \in a.pairs$)**do**

            SetCheckTrueAtom(a, s);

    if($a.fpairs \neq$ *null*) then

        for(all s-atoms $f \in a.fpairs$)**do**

            FatomCheckAtom(f,a.fpairs.bound);

}

Figure 4.1: Pushing an Atom

added literal produced by *pick*. *a.guessed* is changed to false and *not a* pushed on *trail* and another call to expand is made. If after a conflict is found, all literals on *trail* are popped without encountering a guessed atom, there is no answer set compatible with B. The process *pop* is also used to search for additional answer sets when an answer set has been found. When *trail* contains no guessed atoms, all the answer sets compatible with B have been found.

```
procedure AtomBackTrack(a: atom)
{
    if (a.pairs ≠ null) then
        AtomBackTrackSatom(a);
    if (a.fpairs ≠ null) do
        AtomBackTrackFatom(a);
    if (a.Bpos = true) then
        for (all rules r ∈ a.pos) do
            RuleBacktrackFromActive(r);
        for (all rules r ∈ a.neg) do
            RuleBacktrackFromInactive(r);
        a.Bpos := false;
    else
        for (all rules r ∈ a.neg) do
            RuleBacktrackFromActive(r);
        for (all rules r ∈ a.pos) do
            RuleBacktrackFromInactive(r);
        a.Bneg := false;
}
```

Figure 4.2: AtomBackTrack

References

[1] M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R. Watson. An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science-Proceedings of Practical Aspects of Declarative Languages'01*, (2001), 1990:169-183.

[2] C. Baral and M. Gelfond. Reasoning Agents in Dynamic Domains. in *Logic Based Artificial Intelligence* Kluwer, 2000.

[3] T Eiter, W. Faber, N. Leone, G. Pfeifer. Declarative problem solving using the dlv system. In J Minker, editor, *Logic Based Artificial Intelligence*, 79-103 Kluwer, 2000.

[4] M. Gelfond. Representing Knowledge in A-Prolog.

[5] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.,* pages 1070-1080, 1988.

[6] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[7] V. Lifschitz. Action languages, answer sets, and Planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, 357-373, Spring-Verlag, 1999.

[8] J. McCarthy. Elaboration Tolerance. Manuscript,1997.

[9] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375-398, Spring-Verlag, 1999.

[10] I. Niemelä. Logic Programming with stable model semantics as a constraint programming paradigm. In proceedings of the workshop on computational aspects of nonmonotonic reasoning, pp 72-79, Trento, Italy, 1998.

[11] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *5th International Conference, LPNMR'99*, 317-331.

[12] E. Pontelli and O. El-Khatib. Construction and optimization of a parallel engine for answer set programming. In Proceedings of the third symposium on practical aspects of declarative languages, Springer Verlag, pages 288-303, 2001.

[13] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. In Research Report A47, Helsinki University of Technology, August 1997.

[14] P. Simons. Extending the stable model semantics with more expressive rules. In *5th International Conference, LPNMR'99*, 305-316.

[15] P. Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, April, 2000.

[16] T. Soininen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, October, 1998.

[17] T. Soininen. Lparse and lparse user's Manual(draft 0.9). September,2000, http://www.tcs.hut.fi/Software/smodels.