s(ASP) 1.1.0
README

Kyle Marple (kmarple1@hotmail.com)

September 20, 2017

Contents

1 Description

The s(ASP) system is an implementation of the stable model semantics which does not require any form of grounding. Unlike similar systems, it can work with programs which have no finite grounding, as well as accept lists and terms as predicate arguments. Work on s(ASP) is supported by the National Science Foundation under Grant No. 1423419.

For more information, see the paper: Marple, Kyle, Elmer Salazar, and Gopal Gupta. "Computing Stable Models of Normal Logic Programs Without Grounding." arXiv preprint arXiv:1709.00501 (2017) (`https://arxiv.org/abs/1709.00501`).

2 License

The s(ASP) system is distributed under the 3-clause BSD license (a.k.a. BSD-3, Modified BSD License or New BSD License). See the file COPYING for the full text of the license.

3 Package Contents

This README should be part of a distribution containing the following files and directories:

- src/ – Source code directory
- test/ – Test directory containing several sample programs
- CHANGES – Changelog
- COPYING – 3-Clause BSD License
- INSTALL – Dummy file, just directs the reader here
- Makefile – Makefile for systems which support the 'make' command
- README – This file
- README.pdf – A PDF version of this file
- README.tex – A latex version of this file
- winmake.bat – A simple compilation script for Windows

4 Installation

Building s(ASP) requires SWI Prolog to be installed on the local machine. The system has been tested using SWI Prolog version 7.x.x and may be incompatible with older versions. Use of the 64-bit version of SWI Prolog is highly recommended. The 32-bit version restricts memory allocation in such a way that some s(ASP) programs may fail to run due to insufficient stack space. Finally, the compilation methods below assume that it may be invoked using the 'swipl' command.

Prior to compiling s(ASP), you may wish to change the default settings for execution mode, number of answer sets and stack size. This can be done

by editing the file 'src/config.pl'. The various options are explained in the comments of that file.

There are three methods for compiling s(ASP). On systems which support the 'make' command, the included Makefile can be used. On Windows systems without access to the 'make' command, the included 'winmake.bat' may be used. Finally, the following command can be executed from a command prompt in the 'src/' directory:

```
swipl -L0 -G0 -T0 --goal=main --stand_alone=true -o sasp -c main.pl
```

In all three cases, compilation will produce the executable 'sasp', which can then be moved as necessary.

5 Usage

The s(ASP) system accepts modified normal logic programs (see Input Program Format below) and executes them under the stable model semantics. Two modes of execution are supported: automatic and interactive.

Automatic mode will execute programs using either a hard-coded query or an empty query if no hard-coded query is available. The number of stable models computed will be determined by either the default value in 'src/config.pl' or a value given as a command line argument.

Interactive mode works similarly to traditional Prolog systems: the user is provided with a prompt to enter queries. Answers can then be accepted by pressing '.' or rejected by pressing ';'. Pressing 'h' at the prompt will explain both options.

The general format for invoking s(ASP) is:

```
sasp [options] <inputfile(s)>
```

The available options are:

-h, -?, –help	Print this help message and terminate.
-i, –interactive	Run in user / interactive mode.
-a, –auto	Run in automatic mode (no user interaction).
-sN	Compute N answer sets, where N $\geq$ 0. 0 for all. Ignored unless running in automatic mode.
-v, –verbose	Enable verbose progress messages.
-vv, –veryverbose	Enable very verbose progress messages.
-j	Print proof tree for each solution.
-n	Hide goals added to solution by global consistency checks.
-la	Print a separate list of succeeding abducibles with each CHS. List will only be printed if at least one abducible has succeeded.

For example,

```
sasp -i test1.lp
```

will execute the program 'test1.lp' in interactive mode.

6 Input Program Format

The format for s(ASP) programs is that of normal logic programs (NLPs) with a number of optional modifications. These modifications are hard-coded queries, the ability to suppress specific predicates from output, a number of special directive statements, negatively constrained variables and support for arithmetic operations.

6.1 Hard-Coded Queries

Queries may be hard-coded into a program using one of the following two formats:

```
?- goal_1, ..., goal_n.
#compute N { goal_1, ..., goal_n }.
```

Both formats will execute the query 'goal_1, ..., goal_n' when the program is executed in automatic mode. In the second format, N is an integer specifying the number of solutions to compute. A value of 0 indicates that all solutions should be found. Queries using the first format will use the default solution count specified in config.pl. For example, consider the following two queries:

```
?- member(X, [1, 2]).
#compute 3 { member(X, [1,2]) }.
```

The first will compute the default number of solutions, whatever that may be. The second will attempt to compute 3 solutions. As there are only two (assuming a standard definition of member/2), it will find the first two and then fail.

Several things are worth noting: First, the '-s' command line will override any hard-coded solution count. Next, hard-coded queries are ignored when running in interactive mode. Finally, if multiple queries are specified in a single program, all but the last will be silently discarded.

6.2 Hiding Predicates in Output

As the models returned by s(ASP) can be quite large, it is often desirable to omit unwanted predicates from the output. This can be done by appending an underscore ('_') character to the beginning of the predicate name. Such predicates will be executed normally, but neither they nor their negations will be included in any solutions. For example, consider the following program:

```
p.
q :- p.
?- q.
```

When executed, the solution p, q will be printed. However, for the program

```
p.
_q :- p.
?- _q.
```

only p will be printed.

6.3 Directive Statements

In addition to the #compute statement discussed above, two other special statements are supported by s(ASP): #include and #abducible.

The #include statement is used to include additional source files, allowing a program to be broken up across multiple files. Two formats are allowed:

```
#include 'somefile.lp'.
```

and

```
#include('somefile.lp').
```

If the file is in another directory, the relative or absolute path must also be specified:

```
#include 'somedir/somefile.lp'.
#include './somedir/somefile.lp'.
#include '../somedir/somefile.lp'.
#include 'C:\somedirsomefile.lp'. % Windows-specific
```

The #abducible statement is included to simplify the use of s(ASP) for abduction. Abducibles may be specified as follows: #abducible some_goal(X, Y). This simply means that the specified goal may or may not be true. If encountered during execution, it may be "abduced" even if the program contains no rules for the goal.

6.4 Negatively Constrained Variables

In most logic programming systems, variables may be either bound or unbound. In s(ASP), *variables may be either bound or negatively constrained.* A negatively constrained variable is associated with a **prohibited value set**, a list of ground values which the variable cannot be bound to. Unbound variables are treated as a special case of negatively constrained variables in which the prohibited value set is empty.

Values are added to the prohibited value set via **disunification**. Consider the following examples

```
X \= 5, X \= a.
```

Under most logic programming semantics, both of the above statements would fail, since an unbound variable can be bound to any value. However, in s(ASP), both statements will succeed, adding '5' and 'a' to X's prohibited value list. Any later attempt to bind X to one of these values will fail.

6.5 Loop Variables

Loop variables arise from a special case known as an **even loop**, in which a recursive call is encountered with an even, non-zero number of negations between the recursive call and its ancestor. An even loop indicates that every call in the loop may be either true or false, unless some element of the cycle succeeds or is forced to fail via other rules in the program.

If an even loop succeeds with the same negatively constrained variable in both the recursive call and its ancestor, this variable is known as a **loop variable**. In such cases, if the program has at least one stable model, it will have an infinite number of partial stable models. This is because predicates containing the loop variable may be true or false for every value in the loop variable's domain.

The s(ASP) system compactly represents these cases by prefixing loop variables with a '?' in s(ASP)'s output. For example, consider the following program with the query '?- p(X).':

```
p(X) :- not q(X).
q(X) :- not p(X).
```

An infinite number of partial stable models exist for this program, such as:

```
{ p(1), q(2), not p(2), not q(1) }
{ p(1), q(a), q(b), not p(a), not p(b), not q(1) }
```

and so on. However, s(ASP) will produce only the following result:

```
{ p(?X), not q(?X) }
```

This concisely represents that, for each value of ?X, p/1 and q/1 may be either true or false, so long as they are not both true or both false for the same value.

6.6 Arithmetic and Comparisons

The s(ASP) system provides a number of built-in, inline operators, described below.

The two most important operators are unification ('=') and disunification ('\='). For ground values, both operators function identically to their Prolog equivalents. For one ground value and one negatively constrained variable, unification will function as in Prolog, while disunification will add the ground value to the variable's prohibited value list. For two negatively constrained variables, unification will unify the variables and merge their prohibited value lists, while disunification will produce an error (see Section 7).

The remaining arithmetic and comparison operators are detailed in table 1.

7 Restrictions on Input Programs

In addition to the accepted program format, s(ASP) places two limitations on what constitutes a 'legal' program. First, as in Prolog, arithmetic statements must be ground at the time they are evaluated. For example, the statement

Operator	Usage
is	Arithmetic assignment; identical to Prolog operator
=:=	Equality (numbers)
=\=	Inequality (numbers)
<	Less than (numbers)
=<	Less than or equal to (numbers)
>	Greater than (numbers)
>=	Greater than or equal to (numbers)
@<	Less than (terms)
@=<	Less than or equal to (terms)
@>	Greater than (terms)
@>=	Greater than or equal to (terms)
+	Addition
-	Subtraction
*	Multiplication
/	Division
rem	Remainder (integers only)
mod	Modulus (integers only)
<<	Bitwise shift left (integers only): X is 8 << 1 yields X = 16.
>>	Bitwise shift right (integers only): X is 8 >> 1 yields X = 4.
**	Exponent: X is 8**2 yields X = 64.
^	Exponent (identical to **)

Table 1: s(ASP) Arithmetic Operators

```
X is Y + 2.
```

will succeed if Y is bound to a number, but will throw an error otherwise. Second, programs cannot allow success through left recursion. This final restriction is a limitation of the current implementation, and we hope to remove it in a future release.

8 Applications Using s(ASP)

Below are links to projects and papers which use s(ASP). If you would like your project to be listed in future releases, please contact us using the email address at the beginning of this file.

- Chen, Zhuo, Elmer Salazar, Kyle Marple, Gopal Gupta, Lakshman Tamil, Sandeep Das and Alpesh Amin. "Improving Adherence to Heart Failure Management Guidelines Via Abductive Reasoning." Theory and Practice of Logic Programming (2017): 1-16.
- s(ASP) Hackathon (`https://hackai16.devpost.com/submissions`) UT Dallas's AI Society held a 24-hour s(ASP) hackathon on November 5-6, 2016. 18 projects were submitted by teams involving ∼60 UTD students.

- Zhuo Chen, Kyle Marple, Elmer Salazar, Gopal Gupta, Lakshman Tamil: A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns. TPLP 16(5-6): 604-618 (2016) (`https://arxiv.org/abs/1610.08115`)
- Degree Audit Program (`https://gitlab.com/saikiran1096/gradaudit/`) A program to determine if a student has enough credits to graduate and display courses needed fulfill unmet requirements.