Upsh (2.0): A Unix to Prolog Shell *

Nicos Angelopoulos

Department of Computer Science,
University of York, UK.
nicos@cs.york.ac.uk

Abstract. We present Upsh, a lightweight piece of software which enables the execution of Prolog programs from the Unix[1] command prompt. This is achieved either by calling the Upsh executable with a number of arguments or by adding script lines to the source of the Prolog file. We will concentrate more on the former execution mode, since it allows execution of Prolog code without changes to the source code. We detail the arguments our program accepts and explain how these change its behaviour. Upsh runs on three different Prolog systems. The minor limitations on one of them are described.

1 Introduction

Prolog has been used for the last thirty years primarily for academic and research projects. Its relational form provides an intuitive model of encapsulation and a high level of abstraction that favour rapid prototyping of proof-of-concept programs.

Programming in Prolog is relational. A query having some, possibly none, of its arguments fully or partially instantiated with its remaining arguments normally being variables, to be instantiated by the call, is posed against predicate definitions present in memory. This model has been very successful, however, it presents two main limitations. Firstly, it makes it harder to associate Prolog code with identifiable programs having a default behaviour which can be fine-tuned by runtime parameters. Its second limitation, which is to large extend a consequence of the first one, is that it does not provide an effective interface for operating system scripting.

Prolog has a number of features that make it suitable for scripting; most notably its high level of abstraction. This comes at the cost of computational power but it allows for very fast programming and code reuse. As computational power becomes cheaper and programming effort more expensive, Prolog is becoming a more realistic option for scripting.

We present a Prolog program, Upsh, which facilitates communication between the operating system and Prolog programs. An executable saved state is

[*] An earlier version of this work appeared in WLPE'04, St Malo, France.
[1] Trademark of Bell Laboratories

generated from the Upsh sources. This state can be used to execute Prolog programs by commands issued to the operating system shell. The Prolog programs are expected to define an entry-level unary predicate.

Upsh performs transformations which convert operating system friendly arguments, to term structures. These terms are passed as a list of options to the entry-level predicate. Upsh is an attempt to bridge the gap between the pragmatics of operating system programming and the relational form of Prolog. Our software runs on three Prolog systems: SICStus, SWI and Yap. It should be trivial to adjust the software to run on like-minded systems such as CIAO.

This paper is structured as follows. Section 2 provides some preliminary remarks. Section 3 shows how command line arguments are translated to Prolog terms. Section 4 explains the arguments understood by Upsh. Section 5 details where Prolog scripts are looked for and the exit codes. The concluding remarks are in Section 6.

2 Preliminaries

2.1 Overview and Terminology

Although script invocation is supported by most Prologs, there are several drawbacks from the user's perspective: (a) the source code has to be altered; usually adding a couple of lines at the top of the source file, (b) running the code on a different system requires editing of the scripting lines (c) most scripting allowed to-date, limit the callable predicate to something like `main/1`, (d) have poor support for translating operating system arguments to Prolog terms and (e) leave the task of locating the script to the shell call. Upsh attempts to avoid these problems by allowing command line invocation via an executable Prolog state.

In this paper *command line invocation* of Prolog programs. refers to invocation from tcsh-compatible shells. However, the principles apply to any Unix shell with minor modifications. The command line invocation of a Prolog program is facilitated by a saved state generated from the Upsh sources. The process of creating the Upsh state is shown in Fig. 1(a). Once the Upsh source file has been loaded a state is created using the *host engine*'s primitives. There are three logic engines which can host Upsh: SICStus, SWI and Yap. When creating the state, the initial goal is also set to an Upsh defined predicate.

Assuming that the generated state is placed in a directory executed as a shell command. We will refer to *operating system arguments* or simply *arguments* to everything that follows the shell command invoking the executable state.

Once invoked, the state calls an initial predicate. This performs the following: translates the arguments to term structures, loads one or more Prolog source files (also referred to as *scripts*) and calls a single goal of arity 1 or a conjuction of single arity goals. The predicates defining this *entry-goal(s)*, should be present in the scripts. Typically the entry-goal has the same predicate name as the filename minus the '.pl' extension. Fig. 1(b) summarises command line invocation.

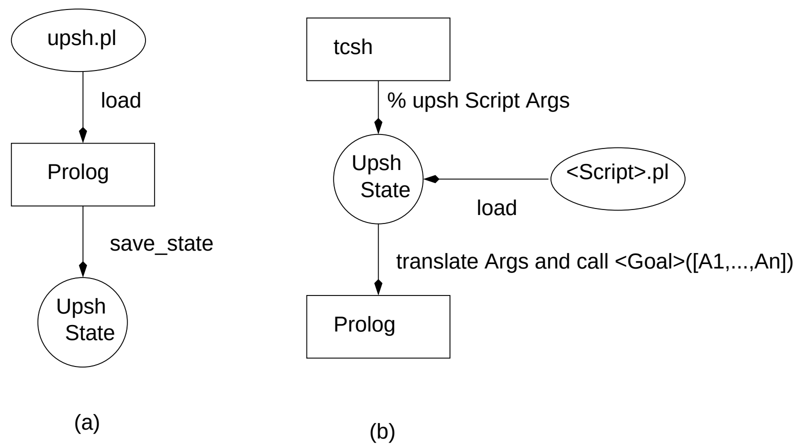

Fig. 1. Creating the Upsh state (a) and command line invocation (b).

The generic form of command line invocation of Upsh is

% `upsh UpshArgs [-] Script ScriptArgs`

`UpshArgs` are one-character long arguments controlling the behaviour of Upsh (these are explained in detail in Section 4). '-' is the separator, which explicitly divides the command line argument to Upsh-specific and script-specific. The first script-specific argument is taken to be the script filename, while the rest are normally translated by *Upsh* to term structures and passed to the script as a list of terms comprising the call parameter to the entry-goal.

We use `<>` around types of objects to which the user is expected to supply a particular value.

2.2 Prolog for scripting

The high level, at which one can program in Prolog, suggests that the language might be an important tool for scripting. Such use has been advocated by the CLIP group at the University of Madrid, e.g. [Cabeza and Hermenegildo, 1997]. Upsh attempts to enhance these possibilities by providing a layer between the operating system and the Prolog code. The main scripting tasks we have had experience with are: building operating system utilities, constructing wrappers for scientific applications, provide auxiliary support to such applications- such as displaying of produced output, and supporting the web interface of a meta-interpreter[2] which uses the Pillow library [Cabeza and Hermenegildo, 1997].

As the cost of computing power reduces while that of programming time rises, we feel that Prolog is becoming more relevant to scripting tasks. Scripts developed in Prolog are faster to compose and debug, easier to maintain and,

[2] `http://www.doc.ic.ac.uk/~nicos/cgi/pfds/pfds.cgi`

in most cases, clearer for other people to understand. The main drawback is the time they take to execute. It is unlikely that Prolog will replace low level languages in scripts that used repeatedly and have a well defined task, e.g. the `grep` command. However, for more specialised tasks the above benefits outweigh the performance penalty.

Of crucial importance to the future of Prolog as a scripting language, is the available of cross-engine libraries which will provide richer building blocks.

2.3 Options versus Arguments

The use of Prolog for scripting depends to a large extent to the provision of an entry-level that is based on options rather than arguments. An option based predicate differs from the usual arguments based predicate, in that a default behaviour is always available even in the absence of any options. The list of options only contains those options for which the user wants to explicitly change their value.

A number of built-in predicates have as part of their arguments a list of options. For instance, see predicate `absolute_file_name/3` in the SICStus manual [SICStus 3.11.1, 2004].

Upsh strengths with regard to scripting and options based code are :

- support of multiple engines
- it does not require changes to the Prolog source code
- provides uniform translation from operating system friendly arguments to Prolog term structures
- its small size
- built-in facilities for locating scripts

3 Argument conversion

Upsh provides a uniform way for converting command line arguments to term structure options passed to the invoked Prolog program. The arguments' syntax is based on special characters that can pass the first parsing by most operating system shells. The arguments are translated to term structures which are easier to manipulate in Prolog. There are three special characters, '=', ',' and ':'. The escape character is \.

In the simplest case, terms are built from '=' argument pairs. An argument pair $Name$=$Value$ will translate to a term $Name(Value)$. Multiple arguments in the term structure are allowed when separated by commas. $Name$= $Value1, \ldots, ValueN$ translate to the term structure $Name(Value1, \ldots, ValueN)$. Spaces in the system arguments are important, since Upsh depends on the executing Prolog to provide the first parse. Thus, no space should appear before '=' or the $Values$ and the separating commas. However, for ease of use spaces are allowed after the special characters.

Arguments not containing an '=' are interpreted as atoms or Prolog variables. An argument is considered to be a Prolog variable if its first character is in upper case. To force the translation of such an argument to an atom, we can prefix the argument with 'atm='. For example, rather=atm=Not will translate to rather('Not').

```
say(Args) :-
    write_args( Args ).

write_args( [] ).
write_args( [H|T] ) :-
    write( H ), nl,
    write_args( T ).
```

Fig. 2. A trivial Prolog script program.

Nested use of '=' is allowed. To signify the end of comma separated values for a nested argument, the outer level comma should be replaced by a ':'. It is thus possible to distinguish between t(n(a,b)) and t(n(a),b). The argument syntax for the former is t=n=a,b while for the latter it is t=n=a:b.

Consider the program in Fig. 2 and recall that the general command line invocation is

% `upsh UpshArgs [-] Script ScriptArgs`

Using the simple script of Table 2 we can demonstrate Upsh's conversion of arguments, with:

% `upsh say fst sec=a1 trd=n1=a2,a3 fo=n2=n3=x::7 atm=V - V h V`

The output from this command is:

```
fst
sec(a1)
trd(n1(a2,a3))
fo(n2(n3(x)),7)
V
_7649
h
_7649
```

This example shows : (a) consecutive '=' are used to create nested terms, (b) 'atm=' prefixes an atomic value that starts with a capital letter, (c) commas are used for multi-argument terms, (d) use of ':' and (e) variables with the same name translate to the same Prolog variable. It is worth noting that '7' is passed to the options as a number and not as an atom, which is the way some engines pass such arguments.

Note that the arguments separator, '-', is used prior to argument 'h' to signal that all the remaining arguments are script specific. Upsh is normally not strict about the use of the separator, however, in this example its use is necessary before ‘h’ because it is a valid Upsh argument. Variables, should also appear after the separator since internally, they can be interpreted as any single letter argument.

Similarly to ‘atm=’, ‘lst=’ and ‘trm=’ are recognised. For example

% `upsh say lst=f,nest=n1,l3=m1:n2,s,lst=t,atm=m:g=i,trm='c-a(b)'`

outputs

```
[f,nest(n1,l3(m1),n2,s,[t,m],g(i,c-a(b)))]
```

The special characters can be escaped when are preceded by ’\’.

4 Upsh Arguments

As can be seen from the help message of Table 1 calls of the Upsh command can be summarised as:
`upsh {h,v}~ {(c),s} [a,b,d,e/1,f/1,l/1,o,p,r,w] - Program [Args]*` Curly brackets separate mutually exclusive arguments. {}~ means these arguments override, i.e. do not actually execute a script, but simply print some information about Upsh and then exit. Argument values within brackets, ‘()’, are the default values for their corresponding set of mutually exclusive argument values. The pair of square brackets hold a number of optional arguments. Arg/1, stands for an argument which expects a parameter, e.g. use ‘e’ in a call as, e=<*ErrorFile*>. []* stands for a number of program specific arguments.

All Upsh arguments are one character long. In this Section, we detail the use of these arguments.

4.1 Invocation

Upsh can be called either from the command line, or from the Prolog script file itself. The main design has been built around the command line invocation, however, scripting is also fully supported. The invocation method should be signalled with one of ‘c’ or ‘s’ arguments, for command line and scripting invocations respectively. These are mutually exclusive arguments, so only one should be given. The default is ‘c’.

When scripting is used, the following two lines should be added to the top of the Prolog script file :

```
#!/bin/sh
exec upsh s - $0 "$@"
```

This requires that Upsh is in a directory in your $PATH environment variable and that the Prolog file has been made executable. When the file is called, the name of the source file will be substituted for $0 and all remaining arguments will be substituted for $@. Alternatively,

```
exec upsh s $0 "$@"
```

can be used. This is more flexible, in that Upsh arguments can be passed at

```
usage: upsh {h,v}~ {(c),s} [a,b,e/1,f/1,d/1,l/1,o,p,r,w] - Program [Args]*.
or
       upsh [c] {(m),n} UArgs [{t,i}= Pn[,Vn] [PnArgs]]*.

        h   : help; overrides,
        v   : version; overrides,

        c   : command line invocation (default),
        s   : script invocation,

        i/2 : provider script Program,
        t/1 : script Program,

        a   : last term will hold list of all read variables (upsh_vs/1).
        b   : be verbose,
        e/1 : errors file (default: std_out stream),
        f   : name of functor for entry-goal (default: <Program>/1,0 then main/1,0),
        d/1 : working directory (default: '.'),
        l/1 : method of loading: the def co(m)pile, co(n)sult, loa(d), (l)ocate or (e)dit
        p   : surpress loading info (when loading Program),
        o   : do not translate Args; e.g. very=nice=tara not to very(nice(tara)),
        r   : be strict about the '-',
        w/1 : when finished do: ring 'bell' wait for 'input', 'both' of the above, or *none*
        y/2 : spy this predicate (Name,Arity),

For scripts add the following lines to the source and do chmod u+x <Program>.pl
#! /bin/sh
exec upsh s - $0 "$@"
```

Table 1. Upsh help message

call time, however, one needs to be more aware about mixing the two types of arguments. Note, that $0 will be recognised correctly as the script's name. This is always taken to be the first non Upsh argument.

Argument 'f=CallName' is used to change the default entry-goal. The entry predicate is always unary. Its predicate name is <*CallName*> and its argument is the list of translated terms that correspond to program-specific arguments of the invoking Upsh call. If this argument is not present `<Program>/1` and `main/1` are looked for, in this order.

Argument 'e=ErrorFile' is used to redirect `user_error` to file <*ErrorFile*>. When absent, this stream is directed to `user_output`. Note, that the identifier to `std_err` is lost, since for some of the supported engines we need to redirect their error stream at invocation. This is the case because on these engines we have no alternative means for avoiding the initial info message.

Alternatively, Upsh can be used with a number of scripts. The general form is:

% `upsh [c] {(m),n} UArgs [{t,i}= Pn[,Vn] [PnArgs]]*`

Each *Pn* is a different script file. In combination with use of variables this presents a powerful way of combining scripts. The *i*/2 option, is for producer scripts. The second argument should be a variable, which will be passed to the script. The intuition is that the script will instantiate this to a term structure which can then be passed to consecutive scripts. In this mode, the entry goal is a composite, conjunctive, one. Each *Pn* contributes a single goal which is formed from its base name and has arity 1. The order remains as the one in the command-line.
For example,

% `upsh i= scripts/tell_story,L t= scripts/say L`

from the distribution's top directory, produces:

```
[the,hello,world,story]
```

Compare this to

% `upsh i=scripts/tell_story,L i=scripts/say,L`

which outputs

```
the
hello
world
story
```

4.2 Loading

The *l*/1 controls the script's loading method. The default value is `m` for compiling the script. Other recognised values are: `n` for consulting, `l` for locating, `d` for loading and `e` for editing. Locating means the location of the script is printed and Upsh exits with loading loads the script and runs nothing. In editing the file is opened in your default editor (pointed-to by $EDITOR shell variable). Loading, consults the script and then exits without running any predicates. In some systems the difference between consulting and compiling is that the latter creates more efficient code, while the former creates more transparent code, e.g. traceable and/or listable.

When using scripts, it is often the case that the user does not want any of the loading information printed by Prolog engines. Upsh provides a uniform way for suppressing such unwanted information with the 'p' argument. We had to implement this feature in a variety of ways for the supported engines. In SICStus we asserted four `portray_message/2` hooks, [SICStus 3.11.1, 2004]. The first argument was always 'informational' while the second arguments used were `imported/3`, `loading/3`, `loaded/6` and `foreign_resource/4`. We simply catch these messages, silently succeeding, thus suppressing any output. In SWI we use the `silent/1` option of `load_files/2`. Finally, Yap provides the informational_messages identifier in prolog_flag/2, see [Costa et al., 2004]. This provides the exact behaviour of 'p' argument.

Argument 'd/1' takes a single parameter. It can be used to change the working directory to the given parameter. The change of directory will take effect before any consultation or other operation, such as creating the <*ErrorFile*> takes place.

4.3 Calling

Argument translation and presentation can be fine-tuned at call time in a number of ways. Argument 'o' suppresses translation of arguments, such as '=' including tokens. For example the argument nice=tara will be left untranslated and passed to the program as the atomic entity: 'nice=tara'. This is also the case for arguments starting with a capital letter; they will be passed as atoms to the program. Note that non-translation applies to all arguments following the separating '-' but has not effect to the ones before the separator. This behaviour ensures translation of Upsh-specific arguments always occurs.

In general, Upsh is not very strict about the use of the separator, '-'. As detailed in Section 2.1 the first use of one-character long arguments that are known to Upsh are interpreted as such. All remaining arguments, apart from the first are translated and passed to the call predicate in the form of a list. The first argument, is taken to be a Prolog program which Upsh will load into memory. Argument 'r' overrides this behaviour by forcing the separation of arguments by '-'. All arguments before '-' are treated as Upsh arguments, while all the ones after the separator are passed to the script. Again, apart from the first one, which is the script source.

In the systems providing codes_to_term/3, it is possible to pass the pairs of variable names and Prolog variables in the script options. The use of argument 'a' adds the option upsh_vs/1 to the script options. Its parameter is a list of $VaribleName = Variable$ pairs. For example consider the following call using the script from Table 2:

```
% upsh a - say T L S T
```

This command's output is:

```
_7054
_7278
_7502
_7054
upsh_vs([[T=_7054],[L=_7278],[S=_7502]])
```

4.4 Modifiers

We have used the term overriding arguments to group the arguments that override, or ignore, all other arguments, by forcing Upsh to write something to the output stream and then exit, without actually running a Prolog script. There are two such arguments: 'h' prints the help message presented in Table 1, while 'v' prints the current version and running engine. For example :

```
% upsh v
Upsh (1:0:0)-yap(4:5:2)-(2004/6/9-15:32:50)
```

In the above, (1:0:0) is the Upsh version, yap(4:5:2) is the engine on which the Upsh state was created and (2004/6/9-15:32:50) is the compilation timestamp.

Argument 'b' switches on the verbose mode. A number of informational messages are printed from within Upsh enabling users to see which arguments are passed to their script along with other details. For example the call

```
% upsh b say tara
```

produces output:

```
all_os_arguments([b,say,tara])
defragmented_os_arguments([b,say,tara])
all_upsh_arguments([b,say,tara])
main_pl_arguments([say,tara])
executing_from_command_line
loading(/home/nicos/pl/upsh/scripts/say)
upsh_calling(say([tara]))
tara
```

`all_upsh_arguments/1` prints the translated form for all the input arguments at the left of any '-'. `main_pl_arguments/1` prints the script file, first element in list, and the arguments that will be passed to it. `executing_from_command_line` informs of the calling method. The following line, gives the full path of the script location. The final line in the presented example, is the output from the script.

When using a script in a terminal whose life-span is identical to that of the script, it is useful to have a uniform way of postponing the end of the process until some input, minimally the enter key, is given by the user. In such or similar scenarios, the input is not necessary part of the script's function but rather a necessity of the method we use to call the script. It is thus useful to incorporate such a simple mechanism into Upsh which is part of the calling mechanism, rather than change all scripts. In this spirit, argument 'w' waits for a single press of the return key, at the end of the script call.

5 Internals

In this section we give details on how Upsh locates the command line scripts, its exit codes and some implementation issues.

5.1 Locating scripts

The simplest method for locating a script is by looking at the current directory. If a matching Prolog file is present in the current directory, then it is selected. Another method is by giving the full or relative path of the file. However, since scripts normally operate on files located in arbitrary places in the filestore, it

```
upsh_look_up( distro, '$HOME/pl/programs' ).
upsh_look_up( recurse, '$HOME/pl/lib/unix' ).
upsh_look_up( say, '$HOME/pl/upsh' ).
upsh_look_up( srm, '$HOME/pl/programs' ).
upsh_look_up( submit, '$HOME/pl/programs' ).
```

Table 2. Example `$HOME/bin/cline_upsh/Lookup.pl` file.

is convenient to uncouple the script location from the location of the files it operates on. Upsh provides two alternative ways of searching for its scripts.

Before compiling Upsh, the definition of `file_search_path/2`, found in `src/upsh.pl` can be edited. Its first argument should be left as is in the source to the value `upsh`. The second argument should point to a directory where scripts can be found. Note, that a number of such directories can be added. At run time, the first directory found to contain a matching Prolog file will be selected. The order is the same as the order the clauses appear in the file. With this method, scripts are collected to specific directories. A benefit of this approach is that it also works for script files which are symbolic links.

Alternatively, script locations can be indexed by script names in the file `\$HOME/bin/cline_upsh/Lookup.pl`. The file should define predicate `upsh_look_up/2`. Each first argument should be the name of a script while the second should the scripts directory location. An example of such a file is shown in Table 2. The main advantage of this approach is that all additions can happen after compiling Upsh and that there is no need to create symbolic links. If another location is preferable for this file, then the source `src/upsh.pl` can be edited and the Upsh state recreated.

Finally, the relative path from current directory can be given. The forward slashes are handled properly by Upsh. Furthermore when 't/1' and 'i/2' are used, the use of a space between the '=' and the script name can be used to assist with shell auto-completion. For example,

% `upsh i= scripts/tell_story,L t= scripts/say L`

from the distribution's top directory, produces:

`[the,hello,world,story]`

5.2 Exit codes

When terminating, Upsh uses `halt/1` with a parameter that reflects the circumstances of termination. The parameter is a small integer that can be picked up from a csh-compatible shell[3] with the use of

% `echo $status`

The codes are

[3]on a sh-compatible shell use: # echo $?

0 everything run to completion successfully
1 an exception was raised while executing the script
2 an error occurred while trying to locate, or load the script file
3 incompatible options were used
4 there was an internal failure. Either
5 no suitable entry-goal found. one of Upsh's own arguments contained unrecognised elements, or other arguments contained syntax errors.

5.3 Implementation Issues

The version can be picked from scripts with
`upsh_flag(upsh_version,UpshVer).`
$UpshVer$ is a composite term: $Ver - Pl - Date$. Ver brakes down to Mj : Mn : Fix, each being an integer. Pl is a term identifying the underlying engine and has the form $Sys(PlMj : PlMn : PlFx)$. $Date$ is composed from $Year/Month/Day - Hour : Min : Sec$ and denotes the time the Upsh state was compiled.

If the entry-goal fails, a warning is printed on 'user_error' and the halt code is set to 1. All predicates in Upsh start with the 'upsh_' prefix. The predicate `upsh_built_call( true )` succeeds at built invocation of Upsh. Useful if you trying to debug the software. Only intended for developers.

To install Upsh change to the distribution directory and use 'upsh_Sys', with Sys to one of 'sicstus' 'swi' or 'yap'. This will create executable ' /bin/upsh'. Multi Upshcan co-exist. Simply rename preceeding executables. For instance from ' /bin/upsh' ' /bin/upsh_swi'. To avoid loading your start-up file on the state creation call to the Prolog engine, use the '-f' flag with any of the three 'make_Sys' shell scripts.

The upsh executable used, can be picked by, either `bb_get( upsh_exec, UExec )` , or in the case of SWI by `flag( upsh_exec, UExec, UExec )` .

6 Conclusions

We have presented a program that helps to bridge the gap between the needs of operating system scripts and those of Prolog programs. Our software can be used to run Prolog scripts in two modes. Command line invocation, in particular, is a new approach providing script calling functionality. In highlight, Upsh (a) does not require changes to the source files, (b) it presents a uniform method of translating between system arguments and Prolog terms, and (c) has flexible mechanisms for locating scripts.

We hope that the presented program will assist in the wider use of Prolog for scripting and in building demonstrative, easy-to-use wrappers around standard Prolog code. In the future it will be interesting to evaluate the impact of Upsh on systems that provide drag and drop interaction.

Upsh runs on three different engines: SICStus, SWI and Yap. Its core functionality should be easy to implement in other Prolog systems, such as CIAOand

GNU Prolog. Upsh is available from `http://www.cs.york.ac.uk/~nicos/sware/upsh/`. It is distributed in source form which experts in other engines can adopt. Our reasons for not supporting more engines is lack of expertise and of available time.

References

[Cabeza and Hermenegildo, 1997] Cabeza, D. and Hermenegildo, M. (1997). Www programming using computational logic sys tems (and the pillow/ciao library). In *Proceedings of the Workshop on Logic Programm ing and the WWW at WWW6*.

[Costa et al., 2004] Costa, V. S., Damas, L., Reis, R., and Azevedo, R. (2004). *Yap 4.5.2. User's Manual*. https://sourceforge.net/projects/yap/.

[SICStus 3.11.1, 2004] SICStus 3.11.1 (2004). *SICStus 3.11.1. User Manual*. http://www.sics.se/isl/sicstus.html.