Exporting Prolog source code

Nicos Angelopoulos
Department of Computing,
Imperial College, London.
nicos@doc.ic.ac.uk

May 13, 2002

Abstract

In this paper we present a simple source code configuration tool. ExLibris operates on libraries and can be used to extract from local libraries all code relevant to a particular project. Our approach is not designed to address problems arising in code production lines, but rather, to support the needs of individual or small teams of researchers who wish to communicate their Prolog programs. In the process, we also wish to accommodate and encourage the writing of reusable code. With consideration to this aim we have set the following objectives: file-based source development, minimal program transformation, simplicity, and minimum number of new primitives.

1 Introduction

Prolog has been around for nearly thirty years. Its ability to survive as a general purpose programming language can be mainly attributed to the fact that it is complimentary to the major players in the field. Without disregard to the many commercial products written in Prolog, the language, arguably, thrives in academic environments, and in particular in AI and proof-of-concept computer science research.

An important element in such projects is that the developers are only expected to write code in a part-time basis within a volatile environment. As a result, programs evolve from few hundred lines to several thousands in an evolutionary manner, that is, without prior overall design of the final product. Indeed, it is seldom the case that an identifiable final product stage is ever reached.

This is contrary to expectations in non-academic settings. As is the fact that sharing and publishing of unfinished source code is desirable. Furthermore tools such as the Unix `make` utility, (Feldman, 1979) which admittedly targets a different set of objectives, requires duplication of work and discourages reusability of Prolog code. In contrast, we present ExLibris which makes use of the directives present in Prolog source files to overcome these problems.

A convenient method for including relatively positioned source code is by using the `library` alias present in most modern Prolog systems. This mechanism is used primarily for system code that implements useful common predicates. For example the `lists` library present in most Prolog systems defines, among others, predicates `member/2` and `append/3`. ExLibris extends the idea by allowing, during project development, access to code from a number of *home* library directories. When one wants to export the project for public use, the source files that are relevant are bundled into a local library directory. The only change required is that the local directory is added as a library directory in the top source files.

This library oriented approach encourages the writing of reusable code. For instance, predicates that accomplish generic tasks should be developed in the home library. Furthermore, it promotes a library oriented way of thinking, where useful code can become independent and in later stages part of the system libraries. For example, the Pillow program (Cabeza and Hermenegildo, 1997) has been incorporated in the current SICStus 3.9.0 release (SICStus 3.9.0, 2002).

Unlike the $DERIVE$ system, (Brereton and Singleton, 1995) we have chosen to use the underlying file store, and to provide in-source support for system-dependent configuration. $DERIVE$ stores predicates in a relational database and uses table attributes to achieve a more holistic approach to Prolog based software engineering.

Dependence on source files mean that in order to accommodate multiple prolog engines and runtime loading we need to introduce some new primitives. In this paper we present a minimum set of such primitives which we believe are interesting in, at least, pointing some of the support needed for such tasks.

ExLibris can be used for configuring both coarse and fine grain libraries. Coarse libraries define many predicates per file, whereas finer grains reduce this to a possibly minimum of one predicate per file. ExLibris depends for the grouping of source files to the primitives provided by the filesystem, that is on the subdirectory relation.

The remaining of this paper is organised as follows. Section 2 deals with some preliminary Prolog definitions that deal with conditional loading and tentative dependencies of source files. Section 3, presents the functionality of ExLibris. Section 4, provides some comments on the features, limitations, and possible future work. Finally, Section 5 serves as the concluding section.

2 Preliminaries

The standard development and configuration phases supported by ExLibris are shown in Fig. 1. Development happens at a project directory which, possibly, contains a local library directory. During development, files in the project directory can use the `library` alias to load any of the following three: system files, that are part of supported prolog engines, home files, that are part of the developer's or the developing team's filespace, and local files, that are within the project's space. ExLibris is a tool that helps to create an export directory

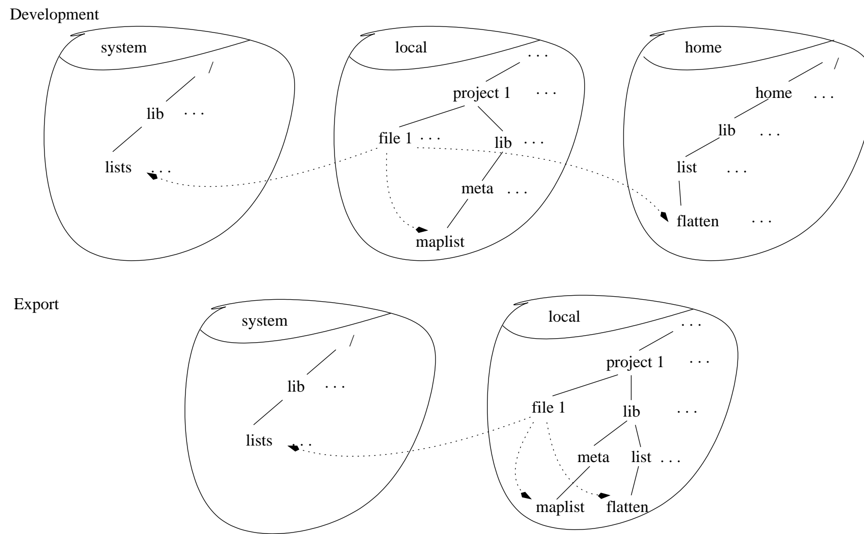

Figure 1: all sorts of libraries

that is independent of home dependencies (as illustrated in the lower part of Fig. 1).

2.1 Conditional load predicates

Since the publication of the Prolog Iso standard (International Standard, 1995; Deransart et al., 1996) the vast majority of systems have strove for compliance. This has made the idea of the same prolog code running on different engine feasible. Still minor differences exist, and it is necessary to take these into account.

The two issues we need to address are, uniform structured prolog identification and conditional loading. These tasks are useful in their own right so, we collect the relevant predicates in the pl library. This has been implemented and tested for SICStus, SWI (Wielemaker, 2002), and Yap (Yap 4.3.20, 2002).

2.1.1 pl/1

Firstly, `pl` defines predicate `pl/1`. Its argument identifies the running prolog system with a compound term, we refer to this term as `pl-term`. The name of the term identifies the prolog system and the term's single argument the version, `pl-version`. The version should be such that the term order imposes the relevant order on the versions. For example the terms for the three most recent versions of the supported systems are: sicstus(3:9:0), swi(5:0:5), and yap(4:3:20).

2.1.2 if_pl/2,3

Files can then be loaded conditionally to the current system. Predicates `if_pl/2,3,4` provide the means for accomplishing this, and can be called as follows:

```
if_pl( +PlTerms, +Call ).
if_pl( +PlName, +PlVersOps, +Call ).
if_pl( +PlName, +PlVersOps, +Call, +ElseCall ).
```

The predicate is quite general since calls *Call* and *ElseCall* can be any callable term. Here we are interested in the cases where `if_pl` is used as a directive and the calls is of the form *LoadCall*(... *Files* ...). *PlTerms* is a single or a list of `pl-terms`. *PlName* is the name part of a `pl-term`. *PlVersOps* is a list of `pl-version` and operator pairs (PlVer-Op). *Files* is a single, or a list of source files. `LoadCall`/*n* is any of the usual load predicate such as `load/1` and `ensure_loaded/1`. An operator is a binary operator that can be applied to two `pl-version` terms. In the `if_pl/2` version *Files* are loaded by calling `LoadCall`, if and only if, run system's `pl-term` unifies with some element of `PlTerms`. In the `if_pl/3` version, *Files* are loaded if and only if (a) the name of the executing Prolog is identical to *PlName* and (b) each `pl-version` satisfies the corresponding Operator when tested against to the executing system's `pl-version`. The call is formed as Op(`load-pl-version`,`run-pl-version`). For example :- if_pl(yap, 4.3.20-`@<`, library('list/flatten')). encountered by any yap system older than 4.3.20 will load file `flatten`. Finally, in `if_pl/4` *ElseCall* is called whenever conditions are not satisfied for executing *Call*.

2.2 Dependent files

Finally, we need to address a discrepancy that arises from loading code at run-time. Unlike when using directives these situations give no easily accessible information about the files a program depends upon. Although it seems useful to have a directive declaring tentative dependencies such feature is not present in any of the discussed systems.

We propose a very simple mechanism facilitated by `may_load/1` directives. :- may_load(+Files) declares that a single or a list of files may be loaded at runtime by the program present in the same source file.

3 Export

Predicate `exlibris/1` is used to create an export directory structure from the developer's source code. The emphasis is placed in integrating relevant parts of private libraries. Its single argument is a list of options. The recognised options are as follows.

`dest`(*Destination*) the destination directory where the exported files will be copied. This should not exist prior to the call. This option does not have a default value.

`source`(*Srcs*) a single file or directory or a list of source files and directories. Each is considered to be either an *entry* level source file, or a directory containing entry level source files. An entry level file is one that a user is expected to load directly. In the case of directories all source files within are considered entry level source files. There is no default value for this option.

`copy`(*Copy*) whether directories containing entry files should also be copied recursively, *Copy* == recursive, or entry files should be copied individually, *Copy* == selective. Default is *Copy* == selective.

`syslib`(*SysLib*) usually a single system library path, but a list of paths can also be given. The provided path should point to the developing Prolog's system library directory. Default is the first directory given as the answer to query ?- library_directory(L).

`homelibs`(*HomeLibs*) a list of private libraries holding source files that are loaded from entry files or their dependents by the `library` alias. The intuition is that during development these directories are defined using `library_directory/1` in entry files or some appropriate start file. The default value is ['`~/prolog/lib`'].

`loclib`(*LocLib*) a path for the local library. This is considered relatively to *Destination*. All referenced files in *HomeLibs* will be copied into *LocLib*. The relative path of any such file from the appropriate *HomeLib* will be recreated within *LocLib*. Note that this may be an existing directory within some source directory. Default value: `lib`.

`pls`(*Pls*) a single or a list of `pl-terms`. Only files pertinent to systems corresponding to these `pl-terms` are copied. These are identified from `if_pl/2‘3` directives as discussed in Section 2.1.2. The default value is for all prologs which is equivalent to `pls(_)`

The exported files are identical to the development ones proviso two transformations. Entry level files loose any `library_directory/1` definition and instead the following lines are added on the top of each such file

```
% Following line added by ExLibris.
:- library_directory( 'RelPathToLocLib' ).
```

When exporting, the value of directory *RelPathToLocLib* is known and it is the path to *LocLib* relative to the particular entry level file. The second transformation is to remove any `if_pl/2‘3` that does not match any of the system `pl-term` in *Pls*.

4 Discussion

Our approach uses the filesystem's directory structure as its medium of grouping predicates at the level of source files. This, supports both fine and coarse

grain groupings. Examples of coarse groupings are the system libraries defining a score of predicates for source file. Whereas, fine grouping would favour single predicate definition per source file or module files exporting a single predicate. However, operations such as moving source files within the home directory structure will need to be accommodated by future tools.

Currently, exlibris runs on SICStus v3.9.0. and Swi v4.0 or later under Unix. Our plans are also to support the Yap and Ciao (Bueno et al., 2002) systems. Yap does not have the `absolute_file_name/3` predicate or the `layout` option for `read_term/2`, while for Ciao we still need to investigate. For SICStus, and since `layout` option only provides the start line of read terms, exlibris requires that if_pl terms are the only terms on the text line in which they appear, and also that there are no new line characters to the end of the term (to the period). Other operating systems may be supported via the support Prolog systems provide for translation of Unix paths to other operating system paths. All code described in this paper can be found at http://www.doc.ic.ac.uk/ nicos/exlibris/

A number of additional tools may be constructed that can help with keeping projects and libraries consistent as well as facilitating library merging. For such tasks, as is also true for other source code manipulation, it will be useful to have a structured form of comments.

In the future we will like to implement non-recursive library copies. That is, the relative path of a home library file is reconstructed into the exported local library directory. This feature is currently not supported because it requires code transformations to a degree greater than we wish the core program to have. One possibility would be to add this as an additional tool that can flatten out any arbitrary library while updating project source files and inter-library dependencies.

5 Conclusions

The first contribution of this paper was to propose simple mechanisms for conditional, depending on the underlying system, loading and execution, and for declaring tentative source file dependencies.

We also provided a straight forward procedure for code configuration and exportation. We have kept core functionalities to a minimum as to encourage simplicity and thus usage. The main contribution of ExLibris is that it encourages development of reusable code.

References

Brereton, P. and Singleton, P. (1995). Deductive software building. In Estublier, J., editor, *Software Configuration Managment. ICSE SCM-4 and SCN-5 Workshops. Selected Papers*, number 1005 in LNCS, pages 81–87. Springer.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., and Puebla,

G. (2002). *Ciao 7.1. User Manual.* The CLIP Group. Technical University of Madrid, Spain. http://www.clip.dia.fi.upm.es/Software/Ciao/.

Cabeza, D. and Hermenegildo, M. (1997). Www programming using computational logic systems (and the pillow/ciao library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6.*

Deransart, P., Ed-Dbali, A., and Cervoni, L. (1996). *Prolog: The Standard.* SpringerVerlag.

Feldman, S. I. (1979). make-a program for maintaining computer programs. *Software - Practise & Experience*, 9:255–265.

International Standard (1995). ISO/IEC 13211-1 (PROLOG: Part 1–general core).

SICStus 3.9.0 (2002). *User Manual.* Swedish Institute of Computer Science, Sweden. http://www.sics.se/isl/sicstus.html.

Wielemaker, J. (2002). *SWI-Prolog 5.0.5. User Manual.* SWI, University of Amsterdam, The Netherlands. http://www.swi-prolog.org.

Yap 4.3.20 (2002). *User Manual.* LIACC/Universidade do Porto and COPPE Sistemas/UFRJ, Portugal. http://www.cos.ufrj.br/ vitor/Yap/.