Prolog Integration and Migration Guide

An application may include plain Prolog files, Prolog modules, and Logtalk objects. This is a perfect valid way of developing a complex application and, in some cases, it might be the most appropriated solution. Modules may be used when a simple encapsulation mechanism is adequate and Logtalk objects may be used when more powerful encapsulation, abstraction, and reuse features are needed. Logtalk supports the compilation of source files containing both plain Prolog and Prolog modules. This guide provides tips for helping integrating and migrating plain Prolog code and Prolog module code to Logtalk. Step-by-step instructions are provided for encapsulating plain Prolog code in objects, converting Prolog modules into objects, and compiling and reusing Prolog modules as objects from inside Logtalk. An interesting application of the techniques described in this guide is a solution for running a Prolog application which uses modules on a Prolog compiler with no module system.

Source files with both Prolog code and Logtalk code

Logtalk source files may contain plain Prolog code intermixed with Logtalk code. The Logtalk compiler just copies the plain Prolog code as-is to the generated Prolog file. With Prolog modules, it is assumed that the module code starts with a `module/1-2` directive and ends at the end of the file. There is no module ending directive which would allowed us define more than one module per file. In fact, most Prolog module systems always define a single module per file. Some of them mandate that the `module/1-2` directive be the first term on a source file. As such, when the Logtalk compiler finds a `module/1-2` directive, it assumes that all code that follows until the end of the file belongs to the module.

Encapsulating plain Prolog code in objects

Most applications consist of several plain Prolog source files, each one defining a few top predicates and auxiliary predicates that are not meant to be directly called by the user. Encapsulating plain Prolog code in objects allows us to make clear the different roles of each predicate, to hide implementation details, and to take advantage of other Logtalk features.

Encapsulating Prolog code using Logtalk objects is easy. First, for each source file, add an opening object directive, `object/1`, to the beginning of the file and an ending object directive, `end_object/0`, to end of the file. Choose an object name that reflects the purpose of source file code. Second, add public predicate directives for the top-level predicates that are used directly by the user or called from other source files. Third, we need to be able to call from inside an object a predicate defined in other source file/object. The easiest solution, which has the advantage of not implying any modification to the predicate clauses, is to use `uses/2` directives. If your Prolog compiler supports cross-referencing tools, you may use them to help you make sure that all calls to predicates on other source files/objects are listed in the `uses/2` directives. Compiling the resulting objects with the Logtalk `portability` flag set to `warning` may help you find calls to predicates defined on other converted source files.

Converting Prolog modules into objects

Converting Prolog modules into objects allows an application to run on a wider range of Prolog compilers, overcoming module compatibility problems. Not all Prolog compilers support a module system. Among those Prolog compilers which support a module system, the lack of standardization leads to several issues, specially with operators and meta-predicates. In addition, the conversion allows you to take advantage of Logtalk more powerful abstraction and reuse mechanisms such as separating interface from implementation.

Converting a Prolog module into a Logtalk object is easy. First, convert the module `module/1` directive into an opening object directive, `object/1`, using the module name as the object name. For `module/2` directives apply the same conversion and convert the list of exported predicates into Logtalk public predicate directives. Second, add a closing object directive, `end_object/0`, at the end of the module code. Third, convert any `export/1` directives into public predicate directives. Fourth, convert any `use_module/2` directives into Logtalk `uses/2` directives. Any `use_module/1` directives are also converted into Logtalk `uses/2` directives but you will need to first find out which predicates your module uses from the specified modules. Fifth, convert any `meta_predicate/1` directives into Logtalk `meta-predicate/1` directives by replacing the module meta-argument indicator, `:`, into the Logtalk meta-predicate indicator, `::`. Arguments which are not meta-arguments are represented by the `*` character. Compiling the resulting objects with the Logtalk `portability` flag set to `warning` may help you find calls to predicates defined on other converted modules.

Compiling Prolog modules as objects

An alternative to convert Prolog modules into objects is to just compile the modules as objects. When possible, this has the advantage of not implying any code changes. You may compile a Prolog module as an object by changing the source file name extension to `.lgt` and then using the `logtalk_load/1-2` and `logtalk_compile/1-2` predicates (set the Logtalk `portability` flag set to `warning` to help you catch any unnoticed cross-module predicate calls). This allows you to reuse existing module code as objects. However, there are some limitations that you should be aware. These limitations are a consequence of the lack of standardization of Prolog module systems. Currently, Logtalk supports the following module directives:

`module/1`
: The module name becomes the object name.

`module/2`
: The module name becomes the object name. The exported predicates become public object predicates.

`use_module/2`
: This directive is compiled as a Logtalk `uses/2` directive in order to ensure correct compilation of the module predicate clauses. Note that the module specified on the directive is not automatically loaded by Logtalk (as it would be when compiling the directive using Prolog instead of Logtalk; the programmer may also want the specified module to be compiled as an object).

`export/1`
: Exported predicates are compiled as public object predicates. The argument must be a predicate indicator (`Functor/Arity`) or a list of predicate indicators.

`meta_predicate/1`
: Module meta-predicates become object meta-predicates. Only predicate arguments marked as `:` are interpreted as meta-arguments. However, note that Prolog module meta-predicates and Logtalk meta-predicates don't share the same exact semantics; check results carefully.

Logtalk supports the use of the *library(name)* notation on the `module/1-2` and `use_module/2` directives (assuming there is an entry for the library on the `logtalk_library_path/2` table).

When compiling modules as objects, you probably don't need event support turned on. Thus, you may want to use the compiler option `events(off)` with the Logtalk compiling and loading built-in methods for a small performance gain for the compiled code.

Current limitations and workarounds

Note that `use_module/1` directives are not currently supported. Therefore, this directives must be converted into `use_module/2` directives by finding which predicates exported by the specified module are imported into the module containing the directive. Automating the conversion would imply loading the module without re-interpreting it as an object, which might not be what the user intended. Nevertheless, finding the names of the imported predicates is easy. First, comment out the `use_module/1` directives and compile the file (making sure that the compiler flag `misspelt` is set to `warning`). Logtalk will print a warning with a list of predicates that are called but never defined. Second, use these list to replace the `use_module/1` directives by `use_module/2` directives. You should then be able to compile the modified Prolog module as an object.

Changing the extension of a module source file to `.lgt` in order to be able to compile it as Logtalk source file is not always feasible. An alternative is to create symbolic links or shortcuts for the module files using `*.lgt` names. In addition, for avoiding conflicts between the Logtalk generated Prolog files and the module files, create the links on a different directory and add a library entry for the directory using the predicate `logtalk_library_path/2`. For example, on a POSIX operating-system with `library/*.pl` module source file names, the links can be easily created by running the following bash shell commands:

```
$ mkdir lgtlib
$ cd lgtlib
$ for i in ../library/*.pl; do ln -sf $i `basename $i .pl`.lgt; done
```

The symbolic links or shortcuts can also be easily created on most operating-systems using the GUI tools.

Dealing with proprietary Prolog directives

Most Prolog compilers define proprietary, non-standard directives that may be used in both plain code and module code. Logtalk will generate compilation errors on source files containing these directives unless you first specify how the directives should be handled. Three actions are possible and can be specified, on a per-directive basis, on the Prolog configuration files: simply ignoring the directive, copying the directive as-is to the generated Prolog files, or rewriting the directive and compiling the resulting one. Each action is specified using, respectively, the predicates: `'$lgt_ignore_pl_directive'/1`, `'$lgt_copy_pl_directive'/1`, and `'$lgt_rewrite_pl_directive'/2`. For example, assume that a given Prolog compiler defines a `comment/2` directive for predicates using the format:

```
:- comment(foo/2, "Brief description of the predicate").
```

We can rewrite this predicate into a Logtalk `info/2` directive by defining a suitable clause for the `'$lgt_rewrite_pl_directive'/2` predicate:

```
'$lgt_rewrite_pl_directive'(comment(F/A, String), info(F/A, [comment is Atom])) :-
    atom_codes(Atom, String).
```

This Logtalk feature can be used to allow compilation of legacy Prolog code without the need of changing the sources. When used, is advisable to set the `portability/1` compiler flag to `warning` in order to more easily identify source files that are likely non-portable across Prolog compilers.

Calling Prolog module predicates using message sending

Logtalk allows you to send a message to a module in order to call one of its predicates. This is usually not advised as it implies a performance penalty when compared to just using the `Module:Call` notation (encapsulating the call between curly brackets when sending the message from inside an object). Note that this only works if there is no object with the same name as the module you are targeting.

This feature is needed to properly support compilation of modules containing `use_module/2` directives as objects. If the modules specified in the `use_module/2` directives are not compiled as objects but are instead loaded as-is by Prolog, the exported predicates would need to be called using the `Module:Call` notation but the converted module will be calling them through message sending. Thus, this feature ensures that, on a module compiled as an object, any predicate calling other module predicates will work as expected either these other modules are loaded as-is or also compiled as objects.

Last updated on: October 26, 2005