Running and debugging Logtalk programs

Running a Logtalk session

We run Logtalk inside a normal Prolog session, after loading the necessary files. Logtalk extends but does not modify your Prolog compiler. We can freely mix Prolog queries with the sending of messages and our programs can be made of both normal Prolog clauses and object definitions.

Starting Logtalk

Depending on your Logtalk installation, you may use a script or a shortcut to start Logtalk with your chosen Prolog compiler. On POSIX operating systems, the scripts should be available from the command-line; scripts are named upon the used Prolog compilers. On Windows, the shortcuts should be available from the Start Menu. If no scripts or shortcuts are available for your installation, operating-system, or Prolog compiler, you can start a Logtalk session by performing the following steps:

1. Start your Prolog compiler.
2. Load the appropriate configuration file for your compiler. Configuration files for most common Prolog compilers can be found in the `configs` subdirectory.
3. Load the Logtalk compiler/runtime files contained in the `compiler` subdirectory.
4. Load the library paths configuration file corresponding to your Logtalk installation contained in the `libpaths` subdirectory.

Note that the configuration files, compiler/runtime files, and library paths file are Prolog source files. The predicate called to load (and compile) them depends on your Prolog compiler. In case of doubt, consult your Prolog compiler reference manual or take a look at the definition of the predicate `'$lgt_load_prolog_code'/1` in the corresponding configuration file.

Most Prolog compilers support automatic loading of an initialization file, which can include the necessary directives to load both the Prolog configuration file and the Logtalk compiler. This feature, when available, allows automatic loading of Logtalk when you start your Prolog compiler.

Compiling and loading your programs

Your programs will be made of source files containing your objects, protocols, and categories. After changing the Prolog working directory to the one containing your files, you can compile them to disk by calling the Logtalk built-in predicate `logtalk_compile/1`:

```
| ?- logtalk_compile([source_file1, source_file2, ...]).
```

This predicate runs the preprocessor on each argument file and, if no fatal errors are found, outputs Prolog source files that can then be consulted or compiled in the usual way by your Prolog compiler. Note that the predicate argument must be either an entity name or a list of entity names.

To compile to disk and also load into memory the source files we can use the Logtalk built-in predicate `logtalk_load/1`:

```
| ?- logtalk_load([source_file1, source_file2, ...]).
```

This predicate works in the same way of the predicate `logtalk_compile/1` but also loads the compiled files to memory.

Both predicates expect a source name name or a list of source name names as an argument. The Logtalk source file name extension, as defined in the configuration file, must be omitted.

If you have more than a few source files then you may want to use a loader helper file containing the calls to the `logtalk_load/1` predicate. Consulting or compiling the loader file will then compile and load all your Logtalk entities into memory (see the next section for details).

Compiler flags

The `logtalk_load/1` and `logtalk_compile/1` always use the current set of default compiler flags as specified in the Logtalk configuration files or changed for the current session using the built-in predicate `set_logtalk_flag/2`. Although the default flag values cover the usual cases, you may want to use a different set of flag values while compiling or loading some of your Logtalk source files. This can be accomplished by using the `logtalk_load/2` or the `logtalk_compile/2` built-in predicates. These two predicates accept a list of flag values affecting how a Logtalk source file is compiled and loaded:

```
| ?- logtalk_compile(Files, Flags).
```

or:

```
| ?- logtalk_load(Files, Flags).
```

In fact, the `logtalk_load/1` and `logtalk_compile/1` predicates are just shortcuts to the extended versions called with the default compiler flag values.

We may also change the default flag values from the ones loaded from the config file by using the `set_logtalk_flag/2` built-in predicate. For example:

```
| ?- set_logtalk_flag(xmldocs, off).
```

The current default flags values can be enumerated using the `current_logtalk_flag/2` built-in predicate:

```
| ?- current_logtalk_flag(xmldocs, Value).

Value = off
yes
```

Lint flags

`unknown(Option)`
: Controls the unknown entity warnings, resulting from loading an entity that references some other entity that is not currently loaded. Possible option values are `warning` (the usual default) and `silent`. Note that these warnings are not always avoidable, specially when using reflective designs of class-based hierarchies.

`misspelt(Option)`
: Controls the misspelt predicate call warnings. A misspelt call is a call to a predicate which is not defined in the object or category containing the call, is not declared as dynamic, and is not a Logtalk/Prolog built-in predicate. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`lgtredef(Option)`
: Controls the Logtalk built-in predicate redefinition warnings. Possible option values are `warning` (the usual default) and `silent`. These warnings are almost always programming errors.

`plredef(Option)`
: Controls the Prolog built-in predicate redefinition warnings. Possible option values are `warning` (can be very verbose if your code redefines a lot of Prolog built-in predicates) and `silent` (the usual default). When running a Logtalk application on several Prolog compilers, is possible to get different sets of warnings due to different sets of built-in predicates implemented by each Prolog compiler.

`portability(Option)`
: Controls the non-ISO specified built-in predicate calls warnings. Possible option values are `warning` and `silent` (the usual default).

`singletons(Option)`
: Controls the singleton variable warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`underscore_vars(Option)`
: Controls the interpretation of variables that start with an underscore (excluding the anonymous variable) that occur once in a term as either don't care variables or singleton variables. Possible option values are `dont_care` and `singletons` (the usual default). Note that, depending on your Prolog compiler, the `read_term/3` built-in predicate may report variables that start with an underscore as singleton variables. There is no standard behavior, hence this option.

Documenting flags

`xmldocs(Option)`
: Controls the automatic generation of documenting files in XML format. Possible option values are `on` (the usual default) and `off`.

`xmlspec(Option)`
: Defines the XML documenting files specification format. Possible option values are `dtd` (for the DTD specification; the usual default) and `xsd` (for the XML Schema specification). Most XSL processors support DTDs but only some of them support XML Schemas.

`xmlsref(Option)`
: Sets the reference to the XML specification file in the automatically generated XML documenting files. The default value is `local`, that is, the reference points to a local DTD or XSD file (respectively, `logtalk.dtd` or `logtalk.xsd`), residing in the same directory as the XML file. Other possible values are `web` (the reference points to an web location, either `http://logtalk.org/xml/1.3/logtalk.dtd` or `http://logtalk.org/xml/1.3/logtalk.xsd`), and `standalone` (no reference to specification files in the XML documenting files). The most appropriated option value depends on the XSL processor you intend to use. Some of them are buggy an may not work with the default option value.

`xslfile(File)`
: Sets the XSLT file to be used with the automatically generated XML documenting files. The default value is `lgtxml.xsl`, which allows the XML files to be viewed by simply opening them with recent versions of web navigators which support XSLT transformations (after copying the `lgtxml.xsl` and of the `logtalk.css` files to the directory containing the XML files).

Other flags

`report(Option)`
: Controls reporting of each compiled or loaded object, category, or protocol (including compilation and loading warnings). Possible option values are `on` (the usual default) and `off` (silent compilation and loading).

`code_prefix(Option)`
: Enables the definition of prefix for all functors of Prolog code generated by the Logtalk compiler. The option value must be an atom; the default value is the empty atom (`''`). Specifying a code prefix provides a way to solve possible conflicts between Logtalk compiled code and other Prolog code. In addition, some Prolog compilers automatically hide predicates whose functor start with a specific prefix such as the character `$`.

`debug(Option)`
: Controls the compilation of source files in debug mode (the Logtalk built-in debugger can only be used with files compiled in this mode). Possible option values are `on` and `off` (the usual default).

`smart_compilation(Option)`
: Controls the use of smart compilation of source files to avoid recompiling files that are unchanged since the last time they are compiled. Possible option values are `on` and `off` (the usual default). This option is only supported in some Prolog compilers. It must not be used when compiling source files in debug mode (see `debug/1` option below).

`events(Option)`
: Allows Logtalk event-driven programming support to be turned off when not needed in order to improve message sending performance. Possible option values are `on` (the usual default) and `off`. Objects (and categories) compiled with this option set to `off` use optimized code for message-sending that does not trigger events. As such, this option can be used on a per-object (or per-category) basis. Note that turning off this option is of no consequence for objects and categories already compiled and loaded.

`hook(Object::Functor)`
: Allows the definition of a compiler hook that is called for each term read form a source file. This option specifies both an object (which can be the pseudo-object `user`) and the functor of a public predicate with two arguments, the first one receiving the term read form the source file and the second one returning a list of terms corresponding to the expansion of the first argument.

Smart compilation of source files

If the Prolog compiler that you are using supports retrieving of file modification dates, then you can turn on smart compilation of source files to avoid recompiling files that have not been modified since last compilation.

Smart compilation of source files is usually off by default. You can turn it on by changing the default flag value in the configuration file, by using the corresponding compiler flag with the compiling and loading built-in predicates, or, for the remaining of a working session, by using the call:

```
| ?- set_logtalk_flag(smart_compilation, on).
```

Some caveats that you should be aware. First, some warnings that might be produced when compiling a source file will not show up if the corresponding object file is up-to-date because the source file is not being (re)compiled. Second, if you are using several Prolog compilers with Logtalk, be sure to perform the first compilation of your source files with smart compilation turned off: the intermediate Prolog files generated by the Logtalk preprocessor may be not compatible across Prolog compilers or even for the same Prolog compiler across operating systems (due to the use of different end-of-line characters).

Using Logtalk for batch processing

If you use Logtalk for batch processing, you probably want to suppress most, if not all, banners, messages, and warnings that are normally printed by the system. To suppress printing of the Logtalk startup banner and default flags, set the option `startup_message` in the config file that you are using to `none`. To suppress printing of compiling and loading messages (including compiling warnings but not compiling error messages), turn off the option `report`.

Debugging Logtalk programs

Logtalk defines a built-in pseudo-object named `debugger`, which implements debugging features similar to those found on most Prolog compilers. However, there are some differences between the usual implementation of Prolog debuggers and the current implementation of the Logtalk debugger that you should be aware. First, unlike some Prolog debuggers, the Logtalk debugger is not implemented as a meta-interpreter. This translates to a different, although similar, set of debugging features with some limitations when compared with some Prolog debuggers. Second, debugging is only possible for objects compiled in debug mode. When compiling an object in debug mode, Logtalk keeps each clause goal in both source form and compiled form in order to allow tracing of the goal execution. Third, implementation of spy points allows the user to specify the execution context for entering the debugger. This feature is a consequence of the encapsulation of predicates inside objects.

Compiling entities in debug mode

Compilation of source files in debug mode is controlled by the compiler flag `debug`. The default value for this flag, usually `off`, is defined in the config files. Its value may be changed at runtime by writing:

```
| ?- set_logtalk_flag(debug, on).

yes
```

In alternative, if we want to compile only some entities in debug mode, we may instead write:

```
| ?- logtalk_load([file1, file2, ...], [debug(on)]).
```

The compiler flag `smart_compilation` is automatically turned off whenever the debug flag is turned on at runtime. This is necessary because debug code would not be generated for files previously compiled in normal mode if there is no changes to the source files. However, note that you should be careful to not turn both flags on at the same time in a config file.

We may check or enumerate, by backtracking, all loaded entities compiled in debug mode as follows:

```
| ?- debugger::debugging(Entity).
```

Logtalk Procedure Box model

Logtalk uses a *Procedure Box model* similar to those found on most Prolog compilers. The traditional Prolog procedure box model uses four ports (*call*, *exit*, *redo*, and *fail*) for describing control flow when a predicate clause is used during program execution:

`call`
: predicate call

`exit`
: success of a predicate call

`redo`
: backtracking into a predicate

`fail`
: failure of a predicate call

Logtalk, as found on some recent Prolog compilers, adds a port for dealing with exceptions thrown when calling a predicate:

`exception`
: predicate call throws an exception

In addition to the ports described above, Logtalk adds two more ports, *fact* and *rule*, which show the result of the unification of a goal with, respectively, a fact and a rule head:

`fact`
: unification success between a goal and a fact

`rule`
: unification success between a goal and a rule head

The user may define for which ports the debugger should pause for user interaction by specifying a list of leashed ports. For example:

```
| ?- debugger::leash([call, exit, fail]).
```

Alternatively, the user may use an atom abbreviation for a pre-defined set of ports. For example:

```
| ?- debugger::leash(loose).
```

The abbreviations defined in Logtalk are similar to those defined on some Prolog compilers:

```
none
    []
loose
    [fact, rule, call]
half
    [fact, rule, call, redo]
tight
    [fact, rule, call, redo, fail, exception]
full
    [fact, rule, call, exit, redo, fail, exception]
```

Defining spy points

Logtalk spy points can be defined by simply stating which predicates should be spied, as in most Prolog debuggers, or by fully specifying the context for activating a spy point.

Defining predicate spy points

Predicate spy points are specified using the method `spy/1`. The argument can be either a predicate indicator (`Functor/Arity`) or a list of predicate indicators. For example:

```
| ?- debugger::spy(foo/2).

Spy points set.
yes

| ?- debugger::spy([foo/4, bar/1]).

Spy points set.
yes
```

Predicate spy points can be removed by using the method `nospy/1`. The argument can be a predicate indicator, a list of predicate indicators, or a non-instantiated variable in which case all predicate spy points will be removed. For example:

```
| ?- debugger::nospy(_).

All matching predicate spy points removed.
yes
```

Defining context spy points

A context spy point is a term describing a message execution context and a goal:

```
(Sender, This, Self, Goal)
```

The debugger is evoked whenever the execution context is true and when the spy point goal unifies with the goal currently being executed. Variable bindings resulting from the unification between the current goal and the goal argument are discarded. The user may establish any number of context spy points as necessary. For example, in order to call the debugger whenever a predicate defined on an object named `foo` is called we may define the following spy point:

```
| ?- debugger::spy(_, foo, _, _).

Spy point set.
yes
```

For example, we can spy all calls to a `foo/2` predicate by setting the condition:

```
| ?- debugger::spy(_, _, _, foo(_, _)).

Spy point set.
yes
```

The method `nospy/4` may be used to remove all matching spy points. For example, the call:

```
| ?- debugger::nospy(_, _, foo, _).

All matching context spy points removed.
yes
```

will remove all context spy points where the value of *Self* matches the name `foo`.

Removing all spy points

We may remove all predicate spy points and all context spy points by using the method `nospyall/0`:

```
| ?- debugger::nospyall.

All predicate spy points removed.
All context spy points removed.
yes
```

Tracing program execution

Logtalk allows tracing of execution for all objects compiled in debug mode. To start the debugger in trace mode, write:

```
| ?- debugger::trace.

yes
```

Note that, when tracing, spy points will be ignored. While tracing, the debugger will pause for user input at each leashed port.

To stop tracing and turning off the debugger, write:

```
| ?- debugger::notrace.

yes
```

Debugging using spy points

Tracing a program execution may generate large amounts of debugging data. Debugging using spy points allows the user to concentrate its attention in specific points of its code. To start a debugging session using spy points, write:

```
| ?- debugger::debug.

yes
```

At the beginning of a port description, the debugger will print a `+` or a `*` before the current goal if there is, respectively, a predicate spy point or a context spy point defined.

To stop the debugger, write:

```
| ?- debugger::nodebug.

yes
```

Note that stopping the debugger does not remove any defined spy points.

Debugging commands

The debugger pauses at leashed posts when tracing or when finding a spy point for user interaction. The commands available are as follows:

`c` — creep
: go on; you may use the spacebar in alternative (but not the return or enter keys)

`l` — leap
: continues execution until the next spy point is found

`s` — skip
: skips debugging for the current goal; only valid at call and redo ports

`f` — fail
: forces backtracking

`n` — nodebug
: turns off debugging

`@` — command
: reads and executes a query

`b` — break
: suspends execution and starts new interpreter; type `end_of_file` to terminate

`a` — abort
: returns to top level interpreter

`d` — display
: writes current goal without using operator notation

`x` — context
: prints execution context

`e` — exception
: prints exception term thrown by the current goal

`=` — debugging
: prints debugging information

`*` — add
: adds a context spy point for the current goal

`+` — add
: adds a predicate spy point for the current goal

`-` — remove
: removes a predicate spy point for the current goal

`h` — help
: prints list of command options; `?` can be used in alternative

Last updated on: October 26, 2005