New Features of Eli4.0

$Revision: 1.9 $

Compiler and Programming Language Group
University of Paderborn, FB 17
33098 Paderborn, FRG

Compiler Tools Group
Electrical and Computer Engineering Department
University of Colorado
Boulder, CO, USA
80309-0425

Program Generation Research Group
Department of Computer Science
James Cook University
Townsville, QLD, 4811 AUSTRALIA

Table of Contents

This document gives information about new facilities available in Eli version 4.0 and those modifications made since the previous distributed Eli version 3.8.3 that might be of general interest. Numerous corrections, improvements, and additions have been made without being described here. They shall just help users to solve their problem without taking notice of Eli's mechanism.

1 Changes to the Eli User-Interface

Invocation of Eli

Eli is built on top of the Odin-System. Up to Version 3.8.3 of the Eli-System, an Odin-Version was integrated in the Eli-System. With the years passing, this odin-system was debugged and adapted to the Eli-System thus making it difficult to integrate new Odin-Versions developed by the author. While Odin is still distributed with Eli, the directories for the two are now completely separated. Eli is actually a set of Odin *packages*.

What does this mean for the Eli-User?

While an '`eli`' invocation script is still provided, all it does is to set an Odin environment variable, `ODINPATH`, to the location of the Eli packages and then invoke the '`odin`' invocation script. Note that the command-line options for this script are different than those older versions of Eli had. Read the manual page for '`odin`' to learn more about the command-line options.

In order to start using Eli, you should set up your account with a few environment-variable settings. First, you must enter the Path to the Odin-System into your Environment-variable '`PATH`'. Since the exact commandline differs in different shells, the following example is valid for usage in '`csh`'-compatible shells:

```
setenv PATH /usr/local/eli/Odin/bin:$PATH
setenv MANPATH /usr/local/eli/Odin/man:$MANPATH
```

If you wish to use the '`odin`' script directly, rather than using the '`eli`' script, you will have to set the `ODINPATH` environment variable to point to the directory where the Eli packages are installed. For example, you might execute the following command before issuing calls to '`odin`'.

```
setenv ODINPATH /usr/local/eli/ELI4.0
```

By default, Odin maintains a cache directory '`.ODIN`' in your home-directory. If you dislike this, you can supply a new definition in the environment-variable '`ODINCACHE`'.

Note that with this new version of Odin you can specify Odin requests to execute on the command-line. Odin will execute the requests and return to the shell. If you do not provide any command-line requests, Odin will come up interactively, which looks like this:

```
Eli Version 4.0 (? for help, ^D to exit) (local)
->
```

Using Eli

In this section, only the main differences between the usage of the Odin-System and the Eli-System up to Version 3.8.3 will be pointed out. For a more complete introduction, please refer to Section 'Overview' in Eli User Interface Reference Manual.

An Odin command that simply specifies an object makes sure that this object is up to date. Up to version 3.8.3 of Eli, such an object was immediately displayed on the Terminal. This is no longer the case. For example, the command `x.specs:exe:warning` will generate a file containing all the warnings and errors while deriving the executable from the given specification. Odin *will not* display the warnings, however.

To display an object, append a `>` to the derivation. A `<` appended to an object will invoke your editor with the filename of the generated object. By appending a `!` and a command, you can invoke a unix-command with the filename of the generated object.

```
x.specs+arg=(i):run     # Make up-to-date
x.specs:parsable<       # To your editor
x.specs>                # To standard output
x.specs:exe>x.exe       # To file x.exe
x.specs:source>src      # To directory src
x.specs:absyntax!more   # Start 'more' with product
```

If an object is executable (e.g. the derivation to '`:help`' yields an executable object), this object is immediately executed upon deriving it. So a derivation `x.specs:exe:help` starts the help-browser with the error- and warning messages occured while deriveing the executable. If you append a '`>`' to the request, you will get the generated executable shellscript displayed.

Another change in the user interface is, that the history mechanism has been dropped. Instead, you can browse in older commandlines using the arrow-keys of your keyboard, See Section 'User Interface' in Quick Reference for Eli 4.0, for further information.

Changed names for derived objects

In the following, you will see a list with the names of legal derivations for Eli Version 3.8.3.

After each derivation, a Eli4.0 substitute will be mentioned. Derivations that have not changed will not be mentioned here.

`:err` — `:err` has been predefined in Odin to mean an object with the raw error output from a derivation (Eli3.8-name was `:.error`) The Eli4.0 name for this is `:error`.

`:warn` — `:warn` has been predefined in Odin to mean an object with the raw error and warning output from a derivation (Eli3.8-name was `:.warning`) The Eli4.0 name for this is `:warning`.

`+arg=(filename):stdout`

Odin has now a more powerful method of executing a generated object. Here, a directory in which the execution should be performed is the main option. The commandline must be supplied in the option `+cmd`. From there, the standard output or the error output can be obtained as objects. A substitute for `x.specs+arg=(in):stdout` is now

```
. +cmd=(x.specs:exe) (in):stdout
```

For further information, See Section 'Supply Command Line Parameters' in Products and Parameters Reference.

`:gencode` — is a list of generated `.c` and `.h` files generated from the processor-specification. To display such a list, use a derivation to `:viewlist` which starts a browser session with the given list. In total, use `:gencode:viewlist` instead of the given Eli3.8-derivation.

This technique also applies to other derivations, for example `:showFe:viewlist`, `:showMe:viewlist`, `:allspecs:viewlist` and `:source:viewlist`.

2 Lido News for Eli 4.0

2.1 Terminals

In previous LIDO versions non-literal terminals were considered to be symbols that may have attributes - several inherited attributes and at most one that is supplied when the node is constructed. Hence, non-literal terminals may have upper symbol computations, and their attributes may be used in adjacent and in remote contexts.

In the new LIDO version a non-literal terminal, like Ident in a rule

```
RULE decl: Declaration ::= Ident ':' Type END;
```

may supply a value to computations associated to the `RULE` context. That value usually describes a property of the corresponding input token, e. g. the encoding of an identifier determined by the scanner and passed through by the parser.

The type of the value supplied by a non-literal terminal is specified by a terminal specification like

```
TERM Ident : int;
```

The above construct can be omitted if the type is `int`, i.e. in all cases where the terminal is provided by an Eli generated scanner. Types other than `int` may occur if tree nodes are created by explicit computations.

The value of a non-literal terminal of a certain context can be used in computations associated to that context, e. g.

```
Declaration.Key = DefineIdn (INCLUDING Root.Env, Ident);
```

There the name of the non-literal terminal stands for its value. If there are several occurrences of a non-literal terminal in a production, their values are distinguished by indexing their names, e. g. `Ident[1]`, `Ident[2]`, ...

The values of non-literal terminals may also be used in symbol computations. There the notation is TERM, TERM[1], TERM[2], ..., e. g. in

```
SYMBOL Use COMPUTE
   SYNT.Key = KeyInEnv (INCLUDING Root.Env, TERM);
END;
```

In order not to immediatly invalidate existing specifications LIGA still accepts most uses of old style terminals and internally transforms them into new style terminals.

2.2 TRANSFER

The TRANSFER construct provided by former LIDO versions is no longer available.

2.3 Separators

The new LIDO version requires semicolons (';') to terminate RULE and SYMBOL specifications and computations. In previous versions of LIDO the ';' after the last specification or computations could be ommited.

2.4 Bottom-Up Evaluation

If computations are to be executed while the input is read they are now to be marked BOTTOM_UP, e.g.

```
printf ("immediate reply\n") BOTTOM_UP;
```

instead of using a LIGAPragma.

Bottom-Up attribute evaluation, i.e. attribute computations during abstract structure tree construction ("parse-time") is no-longer the default strategy used in Eli. To switch to Bottom-Up evaluation it has to be activated in an .ctl specification (ORDER: TREE BOTTOM_UP), See Section 'Order Options' in LCL - Liga Control Language, or requested by a BOTTOM_UP specifier in LIDO See Section 'Computations' in LIDO - Reference Manual.

2.5 CLASS and TREE SYMBOLS

Symbols that describe computational roles (e.g. RangeScope) are now explicitly distinguished from tree grammar symbols by using the keyword CLASS before SYMBOL, e.g.

```
CLASS SYMBOL RangeScope COMPUTE ... END;
```

Symbols that occur in the tree grammar (*tree symbols*) are specified as TREE SYMBOLS:

```
TREE SYMBOL expr COMPUTE ... END;
```

With this extension Liga can then check whether incidentally the name of a tree grammar symbol coincides with a CLASS symbol, that may be obtained from a library module.

Liga will issue warning messages if there is an INHERITS from a tree grammar symbol, or if a CLASS symbol is also a tree grammar symbol. For upward compatibility symbol specifications without TREE or CLASS prefix are still supported.

2.6 LIGAPragma

The LIGAPragma notation of former versions has been substituted by simpler notations (see Section 'Outdated constructs' in LIDO - Reference Manual).

2.7 Terminals in LISTOF productions

In previous LIDO versions terminals were allowed to be used as LISTOF elements, e.g.:

```
RULE: Idents LISTOF Identifier
```

This facility is not supported anymore.

2.8 DEPENDS_ON

An alternative token <- for DEPENDS_ON is accepted.

2.9 LINE, COL, COORDREF

If the source coordinates of contexts are used in computations, the identifiers `LINE`, `COL`, `COORDREF` must occur directly in the Lido text. They may not be introduced by macros defined in a '`.head`' file. As a consequence the library module Message has been removed.

If there is no such coordinate usage in a certain context, that information is not stored in the tree node. The storage needed for the tree is reduced by this means.

2.10 Chain Productions

Chain Productions of the form

```
Production  ::=  SymbName  'IS'  SymbName
```

are no longer valid.

Productions using `::=` now have the same meaning as the former `IS`. The effect of hiding such productions from the parser is achieved by mapping the concrete grammar to the tree grammar.

2.11 Type Definitions

Specifications of the form

```
Specification   ::=     'TYPE' TypeName [Extern] ['LISTEDTO' TypeName]
Extern          ::=     Literal
```

are no longer valid. `TypeNames` are now simply introduced by their use.

2.12 Separated CHAINSTART

Computations of the form

```
Computation     ::=     'CHAINSTART' ChainAttr
```

are no longer valid.

The keyword `CHAINSTART` is now attached to the computation that starts the chain.

2.13 LIDO Tokens

The keyword `STATIC` (equivalent to `COMPUTE`) is no longer valid.

The `:=` token (equivalent to `=`) in computations is no longer valid.

The keyword `CONDITION` in front of plain computations (computations that do not compute an attribute) is now omitted.

3 Changes in Specification Module Library

3.1 General Modifications

The library modules of version 3.6 that have been marked outdated in version 3.8 are removed now. See Section 'Migration of Eli Version 3.6 modules' in Specification Module Library, for migration.

All changes to modules of version 3.8 which may require updates in existing specifications are described in Section 'Migration of Eli Version 3.8 modules' in Specification Module Library.

Modules that do not have a generic parameter are used by their name occurring in a '`.specs`' file, rather than by `:inst` instantiation. If generic parameters are omitted for an instantiation the `:inst` command has still to be used.

All `SYMBOL` roles provided by any library module are specified `CLASS SYMBOL`. By this means accidental name clashes with tree grammar symbols result in warnings. The `INHERITS` construct has to be used to associate a module role to a tree grammar symbol.

Symbol roles that issue a message are separated from roles that compute the condition for such messages. Hence, the message roles can be substituted by individual ones.

3.2 New Modules

Solutions of common type analysis tasks are supported by the following new modules:

The module `BasicType` provides roles for definition and use of typed objects, for type notations and type definitions, Section 'Basic Type Analysis' in Specification Module Library.

The module `Defer` allows to defer association of properties to objects, as required in the presence of type definitions or constant definitions, see Section 'Deferred Property Association' in Specification Module Library.

The module `Operator` supports resolution of overloaded operators using the Oil tool, see Section 'Operator Identification' in Specification Module Library. Its functionality is increased compared to the outdated module AdaptOil.

In the abstract data type library the `List` module has been augmented by a module that can be used if instantiations for several different pointer types are needed, see Section 'Linear Lists of Any Type' in Specification Module Library. In that case a fully typed interface is provided without duplicating the code.

In the output library, two modules for support of pretty printed output have been added. Section 'Pretty Printing' in Specification Module Library, is a module for the support of word wrapping at a specified right margin. Section 'Typesetting for Block Structured Output' in Specification Module Library, is designed to output block-oriented program text. Both modules are very similar and use the new Ptg-Feature of post-processed output.

3.3 Modified Modules

Many details in the name analysis modules are improved. The significant modifications are described in more detail in Section 'Migration of Eli Version 3.8 modules' in Specification Module Library. It is recommended to check the tables of changed modules and modified features for adaption of existing specifications.

Most specifications will be affected by the change of the names `IdDef` to `IdDefScope`, and `IdUse` to `IdUseEnv`, which help to adapt certain specifications to the use of `CLASS SYMBOLS`.

The generic paramerisation of the `PreDefId` module is another prominent change in the name analysis modules.

4 New specification types in PTG

Optional Patterns

In a PTG-Specification rule, optional parts can now be specified. These optional parts will be printed only, if all insertions actually yield output. This can be applied to simplify list construction, see Section 'Optional parts of patterns' in PTG – a Pattern-based Text Generator.

Additionally, a pattern-construction-function yields the special value `PTGNULL`, if its output would be empty. This makes it possible to check a `PTGNode` for empty output,

Postprocessing

PTG now processes it's output by generating applications of a set of output macros. This enables postprocessing the output, e.g. to implement pretty printing or changing output destinations, e.g. to process PTG-Output into an obstack buffer. Applications of this techique are described in the Ptg and the ModLib-documentation, see Section 'Influencing PTG Output' in PTG - a Pattern-based Text Generator.

See Section 'Pretty Printing' in Specification Module Library, for an application for the PTG Postprocessing abilities.

5 New interface to monitoring

Since the last release of Eli the monitoring support has been almost completely rewritten. Apart from internal changes to make the event processing more reliable and a little bit more efficient, the main changes have taken place in the user interface.

The noosa tool which is the main interface to monitoring has a new interface in which all of the monitors are integrated. Previously a new window was created for each monitor. Now they are all handled by the one window, with a common transcript area for output. The only exception is the display of the abstract syntax tree (a new feature) which is performed in a separate window.

Before attempting to use the new monitoring interface it is best to read through the monitoring manual. Also, there is extensive online help which should be consulted while learning the system.

The user interface is somewhat experimental so feedback would be appreciated.

Some work has been done to get the `:mondbx` and `:mongdb` derivations working properly (which they didn't in the previous release). They have been tested with a variety of debuggers but there are some debuggers for which they do not work (e.g., the Solaris non-window version of dbx). They do work with the windowing version of dbx ("debugger" on Solaris) and ups (an X-based debugger). As usual some care is needed to keep track of the current state of execution because the two systems (noosa and the debugger) have different ideas of when the program is running.

6 New command-line processing features

CLP now distinguishes between positional parameters used as input files and other positional parameters. The former must now be designated as input parameters (this is an incompatibility between Eli 4.0 and Eli 3.8). There may only be one of these. Its value is used as the name of the input file (or stdin if the user doesn't specify a value). Command Line Processing.

This new scheme allows processor writers to customise input file handling more closely by using 'plain' positional parameters and handling the input processing themselves.

To avoid clashes with the Eli list modules, CLP has been changed to use those modules instead of its own version (this is an incompatibility between Eli 4.0 and Eli 3.8). Eli 3.8 code that uses the linked lists to access repeated options or positional parameters will have to be changed to work with Eli 4.0. Command Line Processing.

Index