MUltlog 1.17 & iLC 1.2 User Manual

Contents

This manual is available in PDF from the MUltlog website at logic.at/multlog.

MUltlog is a system which takes as input the specification of a finitely-valued first-order logic and produces a sequent calculus, a tableau system, a natural deduction system, and clause formation rules for this logic. All generated rules are optimized regarding their branching degree. The output is in the form of a scientific paper written in LaTeX.

iLC is an editor for Tcl/Tk, which allows to specify many-valued logics for MUltlog in a convenient form.

Further information is available on the project webpage, where you can also find an up-to-date copy of this manual, and example outputs.

1 Requirements

You need the following to run MUltlog:

- MUltlog itself. The source code is available in the Multlog GitHub repository.
- Some standard Prolog system, e.g. SWI-Prolog. Other Prologs might work as well but have not been tested with recent versions of MUltlog.

The output of MUltlog is in the form of a LaTeX paper. To view it properly, you need the typesetting system

- TeX, available from CTAN. For Linux, the TeXLive distribution is particularly convenient to install and is most likely available via your package manager.

MUltlog includes a special editor, iLC, which allows to specify many-valued logics in a convenient, windows-oriented way, instead of typing an ASCII text in a strict syntax. To use this editor you need the script language

- Tcl/Tk (version 7.4/4.0 or later). Many Linux systems include Tcl/Tk by default (check for a program named `wish`).

On Debian/Ubuntu Linux, for instance, you can install the requirements using the command

```
sudo apt install swi-prolog tk texlive-latex-extra
```

2 Installation

2.1 Obtaining MUltlog

Get the newest release of MUltlog by cloning or downloading the Git repository from github.com/rzach/multlog.

2.2 Running the installation script

As of version 1.05, MUltlog comes with an installation script for Linux. Before running the script:

- decide which Prolog to use. The script will look for SWI-Prolog, SICStus, and BinProlog in some standard locations, and suggest the result as a default to the user.
- decide where to put MUltlog. If run as root, the default locations are `/usr/local/bin` for executables, `/usr/local/lib` for library files, and `/usr/share/doc` for documentation. If not run as root, the script will install into `~/.local/bin`, `~/.local/lib`, and `~/.local/doc`. Note that these directories must exist; the script will not try to create them.

To run the installation script, change to the installation directory `multlog` and type

```
./ml_install
```

The script will

- determine the location of some Unix commands
- ask the user for the Prolog to use
- ask the user for the place where to put MUltlog
- generate the deinstallation script `ml_deinstall`
- insert the correct paths in some of MUltlog's files
- copy the MUltlog files in the right places.

In case of problems see the section on troubleshooting below.

Note that the installation procedure puts path information directly into some of MUltlog's files. This means that to install MUltlog somewhere else, you need the original distribution as well as the installation script.

2.3 Deinstallation

Run the script

```
ml_deinstall
```

to remove files installed by `ml_install`. The deinstallation script is located in the same directory as the other MUltlog commands like `lgc2tex`, `lgc2pdf`, ... (`/usr/local/bin` or `~/.local/bin` by default).

3 Using MUltlog

The examples below assume that MUltlog was installed into the standard place `/usr/local/*`, and assumes that the locations of the MUltlog scripts `lgc2tex` and `lgc2pdf` are your command path. If you use different settings, change the examples accordingly.

3.1 Guide for the impatient

- Move to a temporary directory, e.g.,

  ```
  mkdir tmp; cd tmp
  ```

- Get the sample logic from the `doc` directory, e.g.,

  ```
  cp /usr/share/doc/multlog/sample.lgc .
  ```

- Generate the paper in PDF format

  ```
  lgc2pdf sample
  ```

  You should now be able to open `sample.pdf` using the PDF reader of your choice.

- To edit the specification of the logic before generating the paper, type

  ```
  ilc &
  ```

  Select "Open" from the menu "File" and type `sample` as the name of the file to be loaded.

The `examples/` directory of the distribution contains other example specification and configuration files.

3.2 Creating the specification of a logic

You can either use your favourite text editor, or the "interactive Logic Creator" `ilc`.

In the first case specify your many-valued logic in the syntax described in the sample specification `/usr/share/doc/multlog/sample.lgc` and save the result as `<name>.lgc`.

To start `ilc`, type

```
ilc &
```

A window pops up, and you are able to edit a new logic or re-edit an already exisiting one, and to save the result in a textual format suitable for MUltlog. Note that you have to store the logic as `<name>`, the extension `.lgc` being added automatically.

3.3 Creating the paper (PDF)

To obtain the paper corresponding to your logic, type

```
lgc2pdf <name>
```

where `<name>` is the name under which you saved your logic. This invokes MUltlog as well as PDFLaTeX and BibTeX (or alternatively, if `pdflatex` was not found upon installation, LaTeX, BibTeX, `dvips` and `ps2pdf`).

If `<name>.bib` exists, it should contain a bibliography entry with key `ml`, which will be cited as the source for the definition of the logic.

Additionally, all files are deleted except the specification of the logic and the PDF file.

3.4 Creating the paper (LaTeX)

If you are interested in the LaTeX source of the paper, use `lgc2tex` instead of `lgc2pdf`:

```
lgc2tex <name>
```

This will invoke MUltlog, but does neither LaTeXing nor cleaning up. It will produce two files: `<name>.tex` and `<name>.sty`. `<name>.tex` is a template LaTeX file which loads `<name>.sty`. The latter contains the difinitions specific to your logic.

The source will be `<name>.tex` and will require `<name>.sty` and to be compiled. `<name>.sty` contains the definitions produced by MUltlog.

3.5 Creating the paper (DVI)

The command

```
lgc2dvi <name>
```

where `<name>` is the name under which you saved your logic, will produce a DVI file of the paper.

4 Specification of a logic

The directory `/usr/share/doc/multlog` (or whatever you chose) contains a documented example of the configuration file format (as does the `doc` subdirectory of the source distribution itself), `sample.lgc`.

To specify a logic, your specification (`.lgc`) file has to contain the following:

4.1 The name of the logic (mandatory)

Here you specify the name of the logic to be used in the PDF.

Syntax:

```
logic "<logname>".
```

where `<logname>` is a string described by the regular expression RE1

([!#$%&'()*+,./0-9:;<=>?@A-Z[\]^_`a-z{|}~-] | "")*

of up to 40 characters. In other words, the string may consist of any printable ASCII character, where quotes (“) have to be doubled. `<logname>` may contain LaTeX code (e.g., for accented characters), where \ does *not* need to be escaped.

Example:

```
logic "G\""odel".
```

4.2 Truth values (mandatory)

You have to list the truth values of the logic you are defining. The order in which they are listed will be the order used for the presentation of truth tables. Every truth value may appear only once.

Syntax:

```
truth_values { <v 1>, ..., <v n> }.
```

Example:

```
truth_values {f,*,t}.
```

where each of the truth values `<v 1>`, ..., `<v n>` ($n \geq 2$) is described by the regular expression RE2

([a-z][A-Za-z0-9_]* | [-+*^<>=~?@#$&]+ | 0 | [1-9][0-9]*)

The truth values may consist of up to 10 characters. Unless you specify how they should be typeset in the corresponding `.cfg` file, the paper will use the names `<vn>` in italics in the generated PDF.

4.3 Designated truth values (mandatory)

The designated truth values are usually those representing “true”. The particular order of the values is of no significance as indicated by the braces. Every truth value may appear at most once.

Syntax:

```
designated_truth_values { <v 1>, ..., <v n> }.
```

Example:

```
designated_truth_values {t}.
```

where each of the truth values <v 1>, ..., (n>0) is described by the regular expression RE2 above and may consist of up to 10 characters.

The choice of designated truth values has no effect on the generated rules. However, they make a difference to what sequent, initial tableau, or initial clause set has to be used to give a proof of an entailment.

4.4 Orderings of truth values (optional)

By specifying an ordering on truth values, you can declare an operator or quantifier as being the "inf" (greatest lower bound) or "sup" (least upper bound) operator with respect to the ordering.

Syntax:

```
ordering(<ordname>, "<ordspec>").
```

where `<ordname>` is defined according to RE2 above and may consist of up to 10 characters. `<ordspec>` is a string of up to 200 characters satisfying RE1 above. This string is either a single chain, or a set of chains in `{...}`. A chain, in turn, is a seqence of either elements separated by `<`, where each element is either a truth value (as defined by `truth_values`) or itself a set of chains.

In order to avoid ambiguities, spaces may be used to separate the `<` sign from truth values (which may also contain the character `<`).

The semantics of order specifications is as follows:

- Chains like "`a < b < c < d < e`" are interpreted as an abbreviation for "`a < b, b < c, c < d, d < e`".
- Independent chains are collected in sets: "`{a < b, c < d <e, ...}`";
- Sets and chains can be nested. E.g., "`a < {b, c < d} < e`" is the same as "`{a < b, a < c, c < d, b < e, d < e}`".

Let R be the relation defined by this specification. The ordering induced by R is the smallest reflexive, anti-symmetric and transitive relation containing R. Note that truth values with different names are treated as being different from each other. Hence a specification containing `a<b` and `b<a` induces no ordering, since anti-symmetry would imply `a=b`.

Example:

```
ordering(linear, "f < * < t").
```

4.5 Definitions of operators

Optional; but what's a logic without operators?

4.5.1 Mappings

In its simplest and most general form, each operator is specified by its name and the mapping of input to output values. The braces indicate that the order in which the input tuples are assigned output values is of no significance. The definitions should be complete: every k-tuple has to be assigned exactly one value, where $k > 0$ is the arity of the operator. There may be several operators with the same `<opname>` but with different `<arity>`.

Syntax:

```
operator(<opname>/<arity>, mapping { <ass 1>, ..., <
    ass m> }).
```

`<opname>` is defined according to RE2 above and may consist of up to 10 characters. `<arity>` is a non-negative integer. `<ass 1>`, ..., `<ass m>` are assignments of the form

```
( <v 1>, ..., <v k> ) : <v>
```

where `<v 1>`,...,`<v k>`, and `<v>` are truth values. Assignments must be separated by commas. For $k = 0$, the mapping consists of a single truth value.

Example:

```
operator(true /0, mapping { t }).
operator(and  /2, mapping { (t,t): t,
                            (t,*): *,
                            (t,f): f,
                            (*,t): *,
                            (*,*): *,
                            (*,f): f,
                            (f,t): f,
                            (f,*): f,
                            (f,f): f
                          }
```

4.5.2 Tables

Binary operators can be also specified as tables. Since the order of truth values in the table is significant, brackets are used instead of braces.

Syntax:

```
operator(<opname>/2, table [ <v 1>, ..., <v m> ]).
```

<opname> is defined according to RE2 above and may consist of up to 10 characters. <v 1>,...,<v m> are truth values. The number of elements in the table, m, has to be equal to $(n+1)^2 - 1$, where n is the number of different truth values.

Example:

```
operator(and/2, table    [      t, *, f,
                           t,    t, *, f,
                           *,    *, *, f,
                           f,    f, f, f
                         ]
        ).
```

4.5.3 Inf and sup

Operators may also be declared to be the infimum (greatest lower bound) or supremum (least upper bound) with respect to some user-defined ordering.

Syntax:

```
operator(<opname>/<arity>, sup(<ordname>)).
operator(<opname>/<arity>, inf(<ordname>)).
```

Here, <opname> is defined according to RE2 above and may consist of up to 10 characters. <arity> is a non-negative integer greater than one. <ordname> is the name of an appropriate ordering defined by an `ordering` statement.

"sup" stands for the least-upper-bound (= supremum) operation w.r.t. the given ordering. The value of the operator is determined as the least upper bound of the input truth values in the ordering. "inf" stands for the greatest-lower-bound (= infimum) operation w.r.t. the given ordering. The value of the operator is determined as the greatest lower bound of the input truth values in the ordering.

The ordering has to define a unique supremum/infimum for any two truth values.

Examples:

```
operator(and  /2, inf(linear)).
operator(or   /2, sup(linear)).
```

4.6 Definitions of (distribution) quantifiers

Optional.

4.6.1 Mappings

In its simplest and most general form, each quantifier is specified by its name and a mapping assigning a truth value to each non-empty subset of the truth

values. The definitions should be complete: *every* non-empty subset should be assigned exactly one value.

Syntax:

```
quantifier(<quname>, mapping { <ass 1>, ..., <ass m>
    }).
```

<quname> is defined according to RE2 above and may consist of up to 10 characters. <ass 1>,...,<ass m> are assignments of the form { <v 1>, ..., <v k> } : <v>, where <v 1>,...,<v k>, and <v> are truth values. k has to be greater than one.

Example:

```
quantifier(forall, mapping { {t}     : t,
                             {t,*}   : *,
                             {t,f}   : f,
                             {t,*,f}: f,
                             {*}     : *,
                             {*,f}   : f,
                             {f}     : f
                           }
           ).
```

4.6.2 Induced quantifiers

The definition of induced quantifiers in a more comfortable and less error-prone form. Quantifiers can only be induced by operators which are associative, commutative and idempotent.

Syntax:

```
quantifier(<quname>, induced_by <opname>/<arity>).
```

<quname> is defined according to RE2 above and may consist of up to 10 characters. <opname> is defined according to RE2 above and may consist of up to 10 characters, and should be an operator defined as above. <arity> is an integer greater than one.

Example:

```
quantifier(forall, induced_by and/2).
```

4.6.3 Inf and sup

Quantifiers can also be induced by a lub/glb operator.

Syntax:

```
quantifier(<quname>, induced_by <bop>(<ordname>)).
```

`<quname>` is defined according to RE2 above and may consist of up to 10 characters. `<bop>` is either sup or inf. `<ordname>` is the name of an appropriate ordering defined by an "`ordering`"-statement.

Example:

```
quantifier(forall, induced_by inf(linear)).
```

4.7 interactive Logic Creator

`ilc` is a graphical front-end to make creating `.lgc` files a little easier. It requires Tcl/Tk (version 7.4/4.0 or later). To be exact, you only need the executable `wish` (the windowing shell) but not the libraries and none of the extensions.

The program should be rather self-explanatory once you know what goes into the `.lgc` file. You can load and save `.lgc` files from the "File" menu. Before you can specify orderings, operators, and quantifiers, you have to enter the name of the logic and the list of truth values.

5 TeX configuration files

MUltlog will generate a `.tex` and `.sty` file from a given `.lgc` file. If present, it will also use the content of a corresponding `.cfg` file, which can be used to fine-tune the formatted output.

The `.cfg` file for a logic can contain three kinds of declaration:

- `texName(<name>, <definition>)` associates the name `<name>` of a truth value, operator, or quantifier used in the `.lgc` file with LaTeX code used to typeset it. For instance, if `t` is a truth value in the `.lgc` file, then

  ```
  texName(t, "\\mathbf{T}").
  ```

  will result in the truth value `t` be typset as **T**. (Note that `\` have to be doubled.) If the LaTeX replacement is a simple command beginning with a lowercase letter, the quotation marks can be left off, e.g.,

  ```
  texName(and,     \\wedge  ).
  ```

- `texInfix(<op>)` and `texPrefix(<op>)` will cause formulas involving an operator to be set either as infix or prefix, as opposed to the default operator notation. Of course, `texInfix` can only be applied to binary operators, and `texPrefix` only to unary operators. For instance:

  ```
  texInfix(or).
  texPrefix(neg).
  ```

- `texExtra(<macro>, <definition>)` will insert a definition for `\<macro>` with body `<definition>` in the `.sty` file, which will be loaded in the preamble. This can be used to (re-)define any macro used in the LaTeX file. For instance,

  ```
  texExtra("ShortName", "\\textbf{\\L}_3").
  ```

  will define `\ShortName` as `\textbf{\L}_3`, and this macro will be available in the LaTeX file. The LaTeX template file makes use of a number of macros which can be defined in this way:

  - `\Preamble` will be executed in the preamble just before `\begin{document}`. It can be used, e.g., to load packages needed for some of the operator symbols, or to change the document font.
  - `\ShortName` may contain code for an abbreviation or symbol for the logic. `\ShortName` will be called in math mode.
  - `\FullNameOfLogic` is the macro used to insert the name of the logic. By default it will be "`<logname>` logic". Sometimes this doesn't work, so, e.g., you could say:

    ```
    texExtra("FullNameOfLogic", "Halld\\'en's logic
        of nonsense").
    ```

  - `\Intro` will be called (if defined) after the first paragraph of the introduction, and can be used to print a paragraph on the history or motivation of the logic. This paragraph can use `\cite` to generate references to any entries in `<logname>.bib`.
  - `\Semantics` will be called just before the definition of the matrix of the logic. It can be used to print a paragraph, say, about the intuitive interpretation of the truth values, or how the truth functions of the operators are defined (say, on the basis of an ordering).
  - `\Link` will be added as a download link to the citation information at the bottom of the first page.

Using

```
texExtra("Preamble","\\ESequentstrue").
```

you can tell MUltlog to not use the compact representations of sequents, but to explicitly list all components. This will only make sense if the number of truth values is small. In this case, sequents are typeset using the macro `\esequent`. It expects an argument consisting of the components of the sequent, separated by commas. By default,

```
\esequent{{\Gamma_1}, ..., {\Gamma_n}}
```

produces $\Gamma_1 \mid \cdots \mid \Gamma_n$. It can be redefined; e.g., if you have a three-valued logic and want sequents displayed as $\Gamma_1 \Rightarrow \Gamma_2 \mid \Gamma_3$ you could do it using the following `Preamble` declaration:

```
texExtra("Preamble","\\ESequentstrue\\renewcommand{\\
    esequent}[1]{\\sequent##1}\\def\\sequent
    ##1,##2,##3{##1 \\Rightarrow ##2 \\mid ##3}").
```

(Note the double `#`.)

6 Interactive use

The command `multlog` will start Prolog and load the MUltlog source files. This makes it possible to use MUltlog interactively. This feature is expermiental and has not been tested extensively. In particular, it does not yet include detailed error checks.

6.1 Loading and saving logics

Interactive mode allows you to load the specification files of logics and then perform queries and operations on these logics. To load a logic, type, e.g.,

```
?- loadLogic('lukasiewicz.lgc',luk).
```

Here, `?-` is the Prolog prompt; you only enter the text after it. Now the definition of Łukasiewicz logic is available using the ID `luk`.

You can save a logic as an `.lgc` file as well:

```
?- saveLogic('name.lgc',id).
```

6.2 Displaying a logic

To display the truth values and truth tables of your logic, say

```
?- showLogic(luk).
```

This will display the truth tables using the currently selected color scheme. Color schemes are `plain`, `designated`, and `all` and can be set using, e.g., `setColors (luk,plain)`. By default, logics have color scheme `all` which displays different values in different colors, with designated values reversed. This requires an up-to-date version of SWI Prolog. Scheme `plain` just displays all truth values in white, and `designated` in white, but with designated values reversed.

You can also output the truth tables in LaTeX format by saying `showLogic( luk, tex)`. This will require some definitions included in the preamble of your LaTeX document, which can be displayed using `showTexDefs`.

6.3 Operating with formulas and truth tables

Formulas of a logic are built using the operator names in the `.lgc` file, in operator notation. Prolog variables are used for propositional variables. So, e.g., $X \to \neg(X \vee Y)$ would be written as `imp(X, neg(or(X, Y)))`. Instead of variables, you can also put truth values of your logic, e.g., `imp(t, neg(or(*,f )))`. To find the value of this formula:

```
?- hasValue(luk,imp(t, neg(or(*,f))),V).
```

Prolog will display `V = (*)`, i.e., the value is `*`. If you hit space, Prolog will try to find other solutions, and display `false` if no other solutions can be found. In this case, `V=*` is the only solution. However, if the formula contains variables, Prolog will find all solutions. E.g.,

```
?- hasValue(luk,imp(X, neg(or(X,Y))),f).
```

will successively find all values for the variables `X` and `Y` so that the value is `f`:

```
X = t,
Y = f ;
X = t,
Y =  (*) ;
X = Y, Y = t ;
false.
```

The value can itself be a variable. For instance to find a truth value fixed point of $\neg X \vee X$, type.

```
?- hasValue(luk, or(X, neg(X)),X).
```

This will find solutions `*` and `t`.

If you want to know if a formula is designated (or can be made designated), use:

```
?- isDesignated(luk, or(X, neg(X))).
```

This will find the values `t` and `f`. Use `isUnDesignated` instead if you are interested in undesignated values.

To test if a formula is a tautology, say

```
?- isTaut(luk, or(X, neg(X))).
```

This will just produce `false` since $X \vee \neg X$ is not a tautology: if X is `*` the result is `*`, which is not designated:

```
?- isUnDesignated(luk, or(X, neg(X))).
X =  (*)
```

To test for consequence, use

```
?- isConseq(luk, [X, imp(X,Y)], Y).
```

Here, the second argument `[X, imp(X,Y)]` is a *list* of formulas, and since in Łukasiewicz logic, $X, X \to Y \models Y$, this will result in `true`.

You can also test for equivalence of two formulas:

```
?- isEquiv(luk, or(X,Y), luk, imp(imp(X, Y), Y)).
```

Here, the first formula is evaluated according to the operations (truth tables) of the first listed logic, and the second formula according to the operations of the second logic. In this case we use the same logic `luk` for both.

To find formulas with various properties, do the following: `findFmla(logic, F)` will successively find solutions `F` which are formulas of `logic`. The solutions will be ugly, e.g., `F = and(_100, neg(_136))` (`_` followed by a number is Prolog's generic way of naming variables). This can then be combined with other tests, e.g., to find all tautologies, say:

```
?- findFmla(luk,F), isTaut(luk, F).
```

The predicate `findTaut(luk, F)` does the same.

To find a formula equivalent to a given one, use `findEquiv`, e.g.,

```
?- findEquiv(luk, or(X,Y), luk, F).
```

will find all formulas `F` equivalent to $X \vee Y$. The first two are boring—$X \vee Y$ itself and $Y \vee X$—but then it will discover that you can express $X \vee Y$ using $(X \to Y) \to Y$ in Łukasiewicz logic.

To find only formulas not involving $\vee$ here, you can define a second version of Łukasiewicz logic without $\vee$:

```
?- loadLogic('lukasiewicz.lgc',luk2), delOp(luk2,or/2)
   .
```

(`delOp(luk2,op/2)` deletes the 2-place operator `or` from `luk2`.) Now

```
?- findEquiv(luk, or(X,Y), luk2, F).
```

will only find formulas of `luk2` (i.e., formulas not containing `or`) that are equivalent to $X \vee Y$.

You can display a formula in a more readable format using `prettyFmla(F)` or make a pretty copy of a formula using `prettyCopy(F, P)`. For instance, to find and print consequences of logic `l1` that are invalid in `l2` you could say:

```
?- findFmla(l1, and(A,B)), isConseq(l1, [A], B), \+
   isConseq(l2, [A], B),
   prettyCopy((A,B), (Ap, Bp)),
   format('~w entails ~w in ~w but not in ~w~n', [Ap,
      Bp, l1, l2]).
```

Logic `l1` has to include a binary operator—in this case `and`—to find two formulas `A` and `B` with shared variables. It's assumed that the operators of `l1` are also operators of `l2`.

6.4 Operating on logics

If you have two logics loaded or defined, you can have MUltlog define a new logic as the direct product of the two.

```
?- makeProduct(l1, l2, new).
```

The logic `new` is has truth values that are pairs of truth values of the logics `l1` and `l2`, with pairs where both components are designated in `l1` and `l2` being designated in `new`, and operators defined componentwise. This assumes that `l1` and `l2` have the same operators defined.

To make a copy of a logic, say:

```
?- copyLogic(l, new, 'Name').
```

The logic `new` is a copy of logic `l` with name "Name".

You can change the disignated and undesignated values of a logic this way:

```
?- designateTVs(l, [t, f]).
?- undesignateTVs(l, [t, f]).
```

This will change the designated values of Logic `l` to include (or exclude) the values `t` and `f`.

6.5 Congruences and homomorphisms

To find the congruences of a logic, say

```
?- showCong(new).
```

This will look through all partitions of the designated and undesignated values of logic `new` and test if the partition is a congruence. If it is, it will display the partition of truth values and the resulting truth tables, with congruent values colored identically. Each class in the partition is a truth value in the factor logic; truth values are congruent if they are elements of the same class. Equivalent truth values "behave the same" on all operators, e.g., if v and u are equivalent, then $\neg v$ and $\neg u$ are also equivalent. Only "strong" congruences are found, i.e., congruences that respect the designated values (i.e., designated values are equivalent to other designated values, and undesignated values to other undesignated values). (The first congruence found is always the trivial one: every truth value is only equivalent to itself.)

Once you have a congruence, you can define a new logic as the factor logic of the old one by

```
?- makeFactor(logic, part, factor).
```

where `logic` is the ID of the old logic, `part` is the set of sets of truth values that defines the congruence (displayed by `showCong`) and `factor` is the ID of the new logic.

MUltlog can test if two logics are isomorphic, or if there is a homomorphism from one to another:

```
?- isIso(Iso, log1, log2).
?- isHom(Hom, log1, log2).
```

will succeed with `Iso` (`Hom`) bound to a list of pairs of truth values of logics `log1` and `log2` which represents an isomorphism (homomorphism), and fail if no isomorphism exists.

```
?- showHom(Hom, log1, log2).
```

will display the homomorphism `Hom` in a nicer format. To find and show all homomorhisms between `log1` and `log2`, say:

```
?- isHom(Hom, log1, log2), showHom(Hom, log1, log2),
   fail.
```

(The `fail` at the end will automatically find all of them; otherwise you'll have to hit space to backtrack after each.)

7 Creating sequent rule files for MUltseq

The sequent prover MUltseq requires `.msq` files containing the sequent rules for a logic. MUltlog can generate such files from logic specification (`.lgc`) and configuration (`.cfg`) files. Just say

```
lgc2msq <name>
```

8 Troubleshooting

8.1 Installation errors

The installation script may produce the following warnings and errors.

- "Error: `<directory>` does not exist."

  The installation script did not find the directory for executables, library, or documentation (`/usr/local/bin`, `/usr/local/lib`, and `/usr/share/doc` or `~/.local/bin`, `~/.local/lib`, `~./local/doc` by default). Create the directories before running the script or select different directories when prompted.

– "Error: could not find Unix command `<command>`." where `<command>` is one of

```
basename chmod cp dirname false grep mkdir pwd rm
   sed true.
```

The installation script and the scripts for starting MUltlog (`lgc2tex`, `lgc2dvi`, `lgc2pdf`, and `lgc2ilc`) need these Unix commands. The error message means that could not be located, neither on the current command search path nor in the directories `/usr/local/bin`, `/usr/local/sbin`, `/usr/bin`, `/usr/sbin`, `/bin`, or `/sbin`. Locate the directory containing `<command>` and put it on your command search path during installation.

If your Unix system does not have `<command>` at all, submit an issue on https://github.com/rzach/multlog/.

- "Warning: could not find TeX command `<command>`." where `<command>` is one of

  ```
  latex bibtex.
  ```

  The script `lgc2dvi` needs latex and bibtex to produce a DVI-file from the TeX document created by MUltlog. The warning means that `<command>` could not be located, neither on the current command search path nor in the directories

  `/usr/local/bin` `/usr/local/`sbin /usr/bin/usr/sbin/bin/sbin‘

  Check whether TeX is properly installed and put the directory containing `<command>` on your command search path during installation.

- "Warning: couldn't find any PDF converters."

  The script `lgc2pdf` needs either `pdflatex` or `latex`, `dvips` and `ps2pdf` to produce a PDF-file from the TeX document created by MUltlog. The warning means that either `pdflatex` or `dvips` and `ps2pdf` could not be located, neither on the current command search path nor in the directories `/usr/local/bin`, `/usr/local/sbin`, `/usr/bin`, `/usr/sbin`, `/bin`, or `/sbin`. Check whether TeX and Ghostscript are properly installed and put the directory containing the PDF converter on your command search path during installation.

- "Warning: could not find Tcl/Tk command wish."

  The editor `ilc` needs the Tcl/Tk package, in particular the program `wish`. The warning means that `wish` could not be located, neither on the current command search path nor in the in the directories `/usr/local/bin`, `/usr/local/sbin`, `/usr/bin`, `/usr/sbin`, `/bin`, or `/sbin`. Check whether Tcl/Tk is properly installed and put the directory containing `wish` on your command search path during installation.

- "Error: `<command>` does not exist or has no execute permission."

  `<command>` (suggested by the user as Prolog interpreter) does not exist or cannot be executed.

- "Error: `<command>` does not behave like Prolog."

  `<command>` (suggested by the user as Prolog interpreter) exists but fails the test performed by the installation script. This test is a heuristic check whether `<command>` is indeed a Prolog system; more precisely, the output of

  ```
  echo 'f(X,not)=f(ger,Y), print(X),print(Y),halt.'
      | <command>
  ```

  is checked for the string "gernot". If `<command>` is a Prolog system but fails this test, or if it is no Prolog system but passes the test, submit an issue on https://github.com/rzach/multlog/.

8.2 Runtime errors

- Warnings/errors about stack or heap overflows, like `Out of global stack`.

  Such messages indicate that the Prolog system needs more space for the computation than it is currently granted. Check out by which option the space can be increased with your Prolog system. For SWI-Prolog, the command-line option `--stack-limit=2g` will increase the total stack limit from 1GB on 64-bit-architectures (512MB on 32-bit-architectures) to 2GB. There are three ways to tell MUltlog to use this option.

  – *For a single run:* Add the Prolog option as a further argument on the command line.

    ```
    lgc2pdf sample --stack-limit=2g
    ```

  – *Permanently during installation:* Re-install MUltlog. When the installation script asks for the Prolog to be used, type e.g., `/usr/bin/swipl --stack-limit=2g`.

  – *Permanently after installation:* Add the option `--stack-limit=2g` manually, by editing the files

    ```
    /usr/local/bin/lgc2tex
    /usr/local/bin/lgc2dvi
    /usr/local/bin/lgc2pdf
    /usr/local/lib/multlog/ilc/lgc2ilc
    ```

    Near their top there is a line starting with `PROLOG=`. Replace this line, e.g., by

    ```
    PROLOG='/usr/bin/swipl --stack-limit=2g'
    ```

Make sure that the files have still execute permission after saving.

9 About MUltlog

The following people contributed to MUltlog (in alphabetical order):

- Stefan Katzenbeisser and Stefan Kral rewrote the optimization procedure for operators using more efficient data structures.
- Andreas Leitgeb is the author of iLC, the interactive Logic Creator.
- Gernot Salzer wrote the MUltlog kernel and coordinated the project.
- Richard Zach worked on the contents of the MUltlog paper and added the tableaux calculus.