Alg User Manual

Aleš Bizjak[*]   Andrej Bauer[†]

January 21, 2011

Contents

---

[*]`Ales.Bizjak0@gmail.com`, Faculty of Mathematics and Physics, University of Ljubljana
[†]`Andrej.Bauer@andrej.com`, Faculty of Mathematics and Physics, University of Ljubljana

1 Introduction

Alg is a program for enumeration of finite models of single-sorted first-order theories. Such a theory is given by a signature (a list of constants, operations and relations) and axioms expressed in first-order logic. Examples of first-order theories include groups, lattices, rings, fields, posets, graphs, and many others. Alg can do the following:

- list or count all non-isomorphic models of a given theory,

- list or count all non-isomorphic indecomposable[1] models of a given theory.

Currently alg has the following limitations:

- the theory must be single-sorted,

- only unary and binary operations are accepted,

- only (unary) predicates and binary relations are accepted,

- it is assumed that constants denote pairwise distinct elements.

This manual describes how to install and use alg. For a quick start you need Ocaml 3.11 or newer and the menhir parser generator. Compile alg with

```
make
```

and run

```
./alg.native --size 8 theories/unital_commutative_ring.th
```

For usage information type `./alg.native -help` and for examples of theories see the `theories` subdirectory.

Alg is released under the open source simplified BSD License, as detailed in the next section.

2 Copyright and License

Copyright © 2010, Aleš Bizjak and Andrej Bauer

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holder or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

---

[1]A model is indecomposable if it cannot be written as a non-trivial product of two smaller models.

3   Installation

3.1   Downloading alg

Alg is available at `http://hg.andrej.com/alg/`. You have three options:

1. download the ZIP file with source code from

   `http://hg.andrej.com/alg/archive/tip.zip`

2. clone the repository with the Mercurial revision control system:

   `hg clone http://hg.andrej.com/alg/`

3. download a precompiled executable for your architecture, as described at

   `http://math.andrej.com/alg/`

   if one is available.

3.2   Installation for Linux and MacOS

3.2.1   Prerequisites

To compile alg you need Ocaml 3.11 or newer. It is also highly desirable that you have the menhir parser generator and the Make utility. See sections 3.2.6 and 3.2.5 on how to compile without these.

You can get Ocaml and menhir in several ways:

1. On Ubuntu, install the packages `ocaml` and `menhir`:

   `sudo apt-get install ocaml menhir`

   Similar solutions are available on other Linux distributions.

2. On MacOS the easiest way to install Ocaml and menhir is with the macports utility:

   ```
   sudo port install ocaml
   sudo port install caml-menhir
   ```

3. If you have GODI installed then you already have Ocaml. Install menhir with the `godi_console` command, if you do not have it yet.

4. Ocaml is also available from

   `http://caml.inria.fr/`

   and menhir from

   `http://pauillac.inria.fr/~fpottier/menhir/`

3.2.2   Compilation and installation

To compile alg, type `make native` at the command line. If all goes well ocamlbuild will generate a subdirectory `_build` and in it the `alg.native` executable. It will also create a link to `_build/alg.native` from the top directory. To test alg type

   `./alg.native --count --size 8 theories/group.th`

It should tell you within seconds that there are 5 groups of size 8.

We provided only a very rudimentary installation procedure for alg. First edit the `INSTALL_DIR` setting in `Makefile` to set the directory in which alg should be installed, then run

```
sudo make install
```

This will simply copy `_build/alg.native` to `$(INSTALL_DIR)/alg`. You may also wish to stash the `theories` subdirectory somewhere for future reference.

3.2.3   Compiling the manual

You do not have to compile the manual because it is included in `doc/manual.pdf`. Nevertheless, you can recompile it by typing `make doc` or directly by LaTeX-ing `doc/manual.tex`.

3.2.4   Compiling to bytecode

If your version of Ocaml does not compile to native code you can try compiling to bytecode with

```
make byte
```

This will generate a (significantly slower) `alg.byte` executable.

3.2.5   Installation without menhir

If you do not have menhir you can try the following trick:

- move the files `lexer.mll` and `parser.mly` out of the way into some other folder,

- copy the files `nomenhir/lexer.ml` and `nomenhir/parser.ml` to where `lexer.mll` and `parser.mly` used to be,

- compile with `make`.

In other words:

```
\$ mv lexer.mll parser.mly nomenhir
\$ cp nomenhir/lexer.ml nomenhir/parserl.ml .
\$ make
```

3.2.6   Installation without Make

If you do not have the Make utility (how can that be?) you can compile alg directly with ocamlbuild:

```
ocamlbuild -use-menhir alg.native
```

To install alg just copy `_build/alg.native` to `/usr/local/bin/alg` or some other reasonable place.

3.3   Installation for Microsoft Windows

With Ocaml 3.11, menhir and the MinGW or cygwin suite installed compilation should be essentially the same as for Linux and MacOS, as described in section 3.2. Also see sections 3.2.5 and 3.2.6 on how to compile without menhir and Make.

Note that a Windows precompiled executable may be available at `http://math.andrej.com/alg`.

4 Input

An alg input file has extension `.th` and it describes an a first-order theory. The syntax vaguely follows the syntax of the Coq proof assistant. A typical input file might look like this:

```
Theory Group.
Constant 1.
Unary inv.
Binary *.
Axiom unit_left: 1 * x = x.
Axiom unit_right: x * 1 = x.
Axiom inverse_left: x * inv(x) = 1.
Axiom inverse_right: inv(x) * x = 1.
Axiom associativity: (x * y) * z = x * (y * z).
```

The file starts with an optional `Theory` declaration which names the theory, then we have declarations of constants, unary and binary operations, predicates and binary relations (none in the above example), and after that there are the axioms. The precise syntax rules are as follows.

4.1 Comments

Comments are written as in Python, i.e., a comment begins with the `#` symbol and includes everything up to the end of line.

4.2 General syntactic rules

An alg input file consists of a sequence of declarations (`Theory`, `Constant`, `Unary`, `Binary`, `Predicate`, `Relation`) and axioms (`Axiom`, `Theorem`). Each declaration and axiom is terminated with a period.

4.3 The `Theory` keyword

You may give a name to your theory with the declaration

```
Theory theory_name.
```

at the beginning of the input file, possibly preceded by comments and whitespace. The theory name consists of letters, numbers, and the underscore. If you do not provide a theory name, alg will deduce one from the file name.

4.4 Declaration of operations and relations

The declarations

```
Constant c₁ c₂ ... cₖ.
Unary u₁ u₂ ... uₘ.
Binary b₁ b₂ ... bₙ.
Predicate p₁ p₂ ... pₗ.
Relation r₁ r₂ ... rⱼ.
```

are used to declare constants, unary and binary operations, predicates, and binary relations, respectively. You may declare several constants or operations with a single declaration, or one at a time. You may mix declarations and axioms, although it is probably a good idea to declare the constants and operations first.

The keywords `Constants`, `Predicates` and `Relations` are synonyms for their singular counterparts.

A constant may be any string of letters, digits and the underscore character. In particular, a constant may consist just of digits, for example `0` or `1`.

Predicates, relations, unary and binary operations may be strings of letters, digits and the underscore character. For example, if we declare

```
    Unary inv.
    Binary mult.
```

then we can write expressions like `mult(x, inv(y))`. It is even possible to declare operations whose names are strings of digits, for example:

```
    Unary 3 ten.
    Binary +.
    Axiom: 3(3(x)) + x = ten(x).
```

Alternatively, we can use *infix* and *prefix* operators. These follow the Ocaml rules for infix and prefix notation. An operator is a string of symbols

```
    ! $ % & * + - / \ : < = > ? @ \^ | ~
```

where:

- a *prefix operator* is one that starts with `?`, `!` or `~`. It can be used as a unary operation.

- *infix operators* can be used as binary operations and have four levels of precedence, listed from lowest to highest:

  - left-associative operators starting with `=`, `<`, `>`, `|`, `&`, `$`
  - right-associative operators starting with `@` and `^`
  - left-associative operators starting with `+`, `-`, and `\`
  - left-associative operators starting with `*`, `/`, and `%`
  - right-associative operators starting with `**`.

  An operator $\circ$ is *left-associative* if $x \circ y \circ z$ is understood as $(x \circ y) \circ z$, and *right-associative* if $x \circ y \circ z$ is understood as $x \circ (y \circ z)$. If you look at the above list again, you will notice that operators have the expected precedence and associativity. However, if you are unsure about precedence, it is best to use a couple of extra parentheses.

4.5 Axioms

An axiom has the form

```
    Axiom [name]: <formula>.
```

or

```
    Theorem [name]: <formula>.
```

There is no difference between an axiom and a theorem as far as alg is concerned. A good convention is to use `Axiom` for the actual axioms and `Theorem` for statements that are consequences of axioms and are worth including in the theory because they make alg run faster, see Section 7.

The optional *[name]* is a string of of letters, digits and the underscore characters. Alg has no use for names, they are there for the user. The *<formula>* is a first-order formula built from the following logical operations, listed in order of increasing precedence:

$\forall x \,.\, \phi$	is written as	`forall` x`,` ϕ,
$\exists x \,.\, \phi$	is written as	`exists` x`,` ϕ,
$\phi \Leftrightarrow \psi$	is written as	ϕ `<->` ψ or ϕ `<=>` ψ,
$\phi \Rightarrow \psi$	is written as	ϕ `->` ψ or ϕ `=>` ψ,
$\phi \lor \psi$	is written as	ϕ `\/` ψ or ϕ `or` ψ,
$\phi \land \psi$	is written as	ϕ `/\` ψ or ϕ `and` ψ,
$\neg \phi$	is written as	`not` ϕ,
$s = t$	is written as	s `=` t,
$s \neq t$	is written as	s `<>` t or s `!=` t,
$\top$ and $\bot$	are written as	`True` and `False`, respectively.

An iterated quantification $\forall x_1 . \forall x_2 . \cdots \forall x_n . \phi$ may be written as

$$\texttt{forall } x_1 \ x_2 \ \ldots \ x_n \ , \ \phi.$$

and similarly for $\exists$.

Axioms may contain free variables. Thus we can write just

```
Axiom: x + y = y + x.
```

instead of

```
Axiom: forall x y, x + y = y + x.
```

5 Output

5.1 Description of the output

The output of alg is meant to be self-explanatory. Nevertheless, here is what the output consists of:

Title: the name of the theory.

Theory: the input file, which can be suppressed with `--no-source` command-line option.

Models: a list of the models found. This can be suppressed with the `--count` command-line option. Each model has a name *theory_name_n_m* where n is the model size and m is the model sequence number. If a model can be decomposed, a decomposition into indecomposable factors is given.[2] Tables of all the operations and relations are displayed.

Counts: a table showing how many models of each size were found. If more than three sizes were considered, alg also provides a URL to query the counts at `http://oeis.org/`, the On-Line Encyclopedia of Integer Sequences.

5.2 Available output formats

Alg supports several output formats. The default is plain text and it is sent to the screen. You can choose one of the following formats with the `--format` command-line option:

- `text`: the default format is plain text. Actually, it is not entirely plain, as it is also valid Markdown.[3]

- `html`: Hypertext Markup Language, used for web pages.

- `latex`: LaTeX format, suitable for showing multiplication tables in published papers.

- `json`: the JSON format, suitable for futher processing, as explained in Section 5.4.

If you specify an output file with the `--output` option but no `--format`, alg guesses the correct format from the output filename.

5.3 Redirection of output to a file

By default alg prints results on the standard output. You may redirect the output to a file with the `--output` option. If you use `--output` without `--format`, alg tries to guess the output format from the filename extension.

---

[2] Please note that in general such a decomposition is *not* unique.

[3] According to Wikipedia, *Markdown* is a lightweight markup language, originally created by John Gruber and Aaron Swartz to help maximum readability and "publishability" of both its input and output forms.

5.4 Output in JSON format for futher processing

For further processing of output use the JSON output format by specifying `--format json` or output to a file with the `.json` extension.

The JSON output has one of two forms, depending on whether the `--count` option is used. Without it, the output is a list

```
["theory_name", M₁, ..., Mₙ]
```

where $M_1, \ldots, M_n$ are the models found. Each model M_i is given as a dictionary which maps constants, unary operations and binary operations, predicates and relations to the corresponding integers, tables and matrices. For example, the only commutative ring of size 6 with unit is presented as

```
{
  "0" : 0,
  "1" : 4,
  "~" : [0, 2, 1, 3, 5, 4],
  "+" :
    [
      [0, 1, 2, 3, 4, 5],
      [1, 2, 0, 4, 5, 3],
      [2, 0, 1, 5, 3, 4],
      [3, 4, 5, 0, 1, 2],
      [4, 5, 3, 1, 2, 0],
      [5, 3, 4, 2, 0, 1]
    ],
  "*" :
    [
      [0, 0, 0, 0, 0, 0],
      [0, 1, 2, 0, 1, 2],
      [0, 2, 1, 0, 2, 1],
      [0, 0, 0, 3, 3, 3],
      [0, 1, 2, 3, 4, 5],
      [0, 2, 1, 3, 5, 4]
    ]
}
```

The underlying set of the ring is $\{0, 1, 2, 3, 4, 5\}$, the neutral elements for addition and multiplication are 0 and 4, respectively, and the operations are given as lookup tables, for example, ˜$1 = 2$ and $4 * 3 = 3$.

The `--count` option gives JSON output

```
["theory_name", [[i₁,k₁], ..., [iₙ,kₙ]]]
```

which is read as saying that there are k_j models of size i_j.

In Python you can import JSON data from a file `mystuff.json` like this:

```
import json
with open('mystuff.json','r') as f:
    mystuff = json.load(f)
```

In Mathematica just use the `Import["mystuff.json"]` command.

6 Command-line Options

Alg is used as

```
alg --size <sizes> [options] <theory.th>
```

where *theory.th* is the input file, and the options are:

8

--size *<sizes>* A comma-separated list of sizes that alg should consider. You can also specify a size interval of the form `m-`*n*. For example, `1,2,5-8` would mean that we consider sizes 1, 2, 5, 6, 7, 8.

--count Do not print out the models, just report the counts.

--format *<format>* Output in the given format. Supported formats are `text`, `html`, and `latex`.

--indecomposable Output only indecomposable models, i.e., those that are not products of smaller models.

--axiom Add an extra axiom to the theory. Use the option several times to add several axioms. Note that on a command-line the axiom should be enclosed in quotes.

--no-products Do not try to generate models as products of smaller models. Use this option if you know that a model cannot be a product of smaller ones. For example, a field can never be a product of two fields. The option is assumed if the theory contains predicates or relations. If all of your axioms are equations, then you should *not* use this option.

--no-source Do not include the theory in the output.

--output *<filename>* Output to the given file rather than to screen.

--paranoid Perform additional checks to verify that the models really satisfy the axioms. Use this option if you think there is a bug in alg.

--sat Use the alternative algorithm based on satisfying the axioms rather than intelligent filling-in of multiplication tables. It works well with theories that have predicates and relations. For theories with unary and binary operations the standard algorithm typically performs better.

--version Print out version information and exit.

--help Print help.

7 How to use alg efficiently

You should always keep in mind the fact that alg performs a brute force search with a few optimizations. In the worst case its running time is doubly exponential in the size of the models because there are n^{n^2} tables for a binary operation on a set of size n.

Alg contains two diffrent algorithms, the *default* and the *alternative* one. As a rule of thumb you should use the default algorithm, unless the signature of your theory consists mostly of predicates and relations. In any case, since the performance is not easily predicted, you should try both algorithms and experiment by rewriting the theory in various equivalent ways.

7.1 The default algorithm

The default alg algorithm is optimized for *equational* theories, i.e., those whose axioms are equations (semigroups, monoids, groups, rings, lattices, etc., but *not* integral domains and fields). Alg takes advantage of commutativity, associativity and idempotent laws, and to a smaller extent of other kinds of equational laws, such as absorption and distributivity.

Alg performs few optimizations based on non-equational axioms. Thus you should try to reduce their number. Furthermore, in non-equational axioms the quantifiers should always be pushed inside as much as possible. For example, instead of

```
Axiom: forall x, exists y, x <> 0 -> x * y = 1.
```

you should write

```
Axiom: forall x, x <> 0 -> exists y, x * y = 1.
```

The best kind of axioms are those that allow alg to immediately fill in a whole column or row. Typically these are axioms about neutral elements, such as $1 \cdot x = x$ and $0 + x = x$. As a rule of thumb, every such axiom will increase the maximum manageable size by one.

In general alg performs better if it is given more axioms and theorems, because each additional statement cuts down the possibilities. Thus you *should* include theorems which already follow from other axioms. For example:

- state *both* $1 \cdot x = x$ and $x \cdot 1 = x$, even if one of them follows from the other,

- more generally, state all versions of a symmetric equation, even if they all follow from one of them,

- state laws like $0 \cdot x = 0$ (and also $x \cdot 0 = x$), even if they follow from other axioms.

You should declare as many constants and as few operations as possible. A typical example is the theory of lattices. For *finite* structures the following are equivalent theories:

- a lattice with operations $\wedge$, $\vee$,

- a bounded lattice with operations $\wedge$, $\vee$ and constants 0, 1,

- a $\vee$-semilattice with operation $\vee$ and constant 0,

- a bounded $\vee$-semilattice with operation $\vee$ and constants 0, 1.

The best choice is the last one because it has just fewest operation and most constants. Indeed, figuring out that there are 53 lattices of size 7 takes 250 times longer with the theory of a lattice than with the theory of a bounded $\vee$-semilattice.

Lastly, we should mention that alg generates tables in the order in which the operations are declared. Sometimes it is much easier to generate tables for one operation than another, so you should experiment by switching the order of declarations. For example, the theory of a ring works much faster if addition $+$ is declared before multiplication $\times$.

7.2   The alternative algorithm

The alternative algorithm transforms the given axioms into constraints that need to be satisfied. It then solves the constraints in all possible ways by intelligent backtracking. The general rules for efficient use of the alternative algorithm are the same as those for the default one.

It turns out that the alternative algorithm generally performs worse than the default one when the theory consists mostly of operations. However, the alternative algorithm usually outperforms the default one for theories which constist mostly of relations.

8   Support and bug reports

Please send bug reports and whishes to the authors: Andrej Bauer (`Andrej.Bauer@andrej.com`) and Ale Bizjak (`Ales.Bizjak0@gmail.com`). If you report a bug, please send us the exact version number of alg, as printed out with the `--version` command-line option.

The Mathematics and Computation blog at `http://math.andrej.com` may contain further information and discussion about alg.