Redlog User Manual

A REDUCE Logic Package
Edition 2.0, for REDLOG Version 2.0 (REDUCE 3.7)
15 April 1999

by Andreas Dolzmann and Thomas Sturm

Acknowledgments

We acknowledge with pleasure the superb support by Herbert Melenk and Winfried Neun of the Konrad-Zuse-Zentrum fuer Informationstechnik Berlin, Germany, during this project. Furthermore, we wish to mention numerous valuable mathematical ideas contributed by Volker Weispfenning, University of Passau, Germany.

Redlog Home Page

There is a REDLOG home page maintained at
`http://www.fmi.uni-passau.de/~redlog/`.
It contains information on REDLOG, online versions of publications, and a collection of examples that can be computed with REDLOG. This site will also be used for providing update versions of REDLOG.

Support

For mathematical and technical support, contact
`redlog@fmi.uni-passau.de`.

Bug Reports and Comments

Send bug reports and comments to
`redlog@fmi.uni-passau.de`.
Any hint or suggestion is welcome. In particular, we are interested in practical problems to the solution of which REDLOG has contributed.

1 Introduction

REDLOG stands for REDUCE *logic system*. It provides an extension of the computer algebra system REDUCE to a *computer logic system* implementing symbolic algorithms on first-order formulas wrt. temporarily fixed first-order languages and theories.

This document serves as a user guide describing the usage of REDLOG from the algebraic mode of REDUCE. For a detailed description of the system design see [DS97a].

An overview on some of the application areas of REDLOG is given in [DSW98]. See also Chapter 7 [References], page 32 for articles on REDLOG applications.

1.1 Contexts

REDLOG is designed for working with several *languages* and *theories* in the sense of first-order logic. Both a language and a theory make up a context. In addition, a context determines the internal representation of terms. There are the following contexts available:

OFSF
: OF stands for *ordered fields*, which is a little imprecise. The quantifier elimination actually requires the more restricted class of *real closed fields*, while most of the tool-like algorithms are generally correct for ordered fields. One usually has in mind real numbers with ordering when using OFSF.

DVFSF
: *Discretely valued fields*. This is for computing with formulas over classes of p-adic valued extension fields of the rationals, usually the fields of p-adic numbers for some prime p.

ACFSF
: *Algebraically closed fields* such as the complex numbers.

The trailing "-SF" stands for *standard form*, which is the representation chosen for the terms within the implementation. See Section 2.2 [Context Selection], page 5 for details on selecting REDLOG contexts.

1.2 Overview

REDLOG origins from the implementation of quantifier elimination procedures. Successfully applying such methods to both academic and real-world problems, the authors have developed over the time a large set of formula-manipulating tools, many of which are meanwhile interesting in their own right:

- Numerous tools for comfortably inputing, decomposing, and analyzing formulas. This includes, for instance, the construction of systematic formulas via `for`-loops, and the extension of built-in commands such as `sub` or `part`. See Chapter 3 [Format and Handling of Formulas], page 7.

- Several techniques for the *simplification* of formulas. The simplifiers do not only operate on the boolean structure of the formulas but also discover algebraic relationships. For this purpose, we make use of advanced algebraic concepts such as Groebner basis computations. For the notion of simplification and a detailed description of the implemented techniques for the contexts OFSF and ACFSF see [DS97]. For the DVFSF context see [DS99]. See Chapter 4 [Simplification], page 14.
- Various *normal form computations*. The CNF/DNF computation includes both boolean and algebraic simplification strategies [DS97]. The *prenex normal form* computation minimizes the number of quantifier changes. See Chapter 5 [Normal Forms], page 25.
- *Quantifier elimination* computes quantifier-free equivalents for given first-order formulas.

  For OFSF and DVFSF we use a technique based on elimination set ideas [Wei88]. The OFSF implementation is restricted to at most quadratic occurrences of the quantified variables, but includes numerous heuristic strategies for coping with higher degrees. See [LW93], [Wei97] for details on the method. The DVFSF implementation is restricted to formulas that are linear in the quantified variables. The method is described in detail in [Stu98].

  The ACFSF quantifier elimination is based on *comprehensive Groebner basis* computation. There are no degree restrictions for this context [Wei92]. See Section 6.1 [Quantifier Elimination], page 27.
- The contexts OFSF and ACFSF allow a variant of quantifier elimination called *generic quantifier elimination* [DSW98]. There are certain non-degeneracy assumptions made on the parameters, which considerably speeds up the elimination.

  For geometric theorem proving it has turned out that these assumptions correspond to reasonable geometric non-degeneracy conditions [DSW98]. Generic quantifier elimination has turned out useful also for physical applications such as network analysis [Stu97]. There is no generic quantifier elimination available for DVFSF. See Section 6.2 [Generic Quantifier Elimination], page 29.
- The contexts OFSF and DVFSF provide variants of (generic) quantifier elimination that additionally compute *answers* such as satisfying sample points for existentially quantified formulas. This has been referred to as the `"extended quantifier elimination problem"` [Wei97a]. See Section 6.1 [Quantifier Elimination], page 27.
- OFSF includes linear *optimization* techniques based on quantifier elimination [Wei94a]. See Section 6.3 [Linear Optimization], page 30.

1.3 Conventions

To avoid ambiguities with other packages, all REDLOG functions and switches are prefixed by `"rl"`. The remaining part of the name is explained by the first sentence of the documentation of the single functions and switches.

Some of the numerous switches of REDLOG have been introduced only for finding the right fine tuning of the functions, or for internal experiments. They should not be changed anymore, except for in very special situations. For an easier orientation the switches are divided into three categories for documentation:

Switch
: This is an ordinary switch, which usually selects strategies appropriate for a particular input, or determines the required trade-off between computation-speed and quality of the result.

Advanced Switch
: They are used like ordinary switches. You need, however, a good knowledge about the underlying algorithms for making use of it.

Fix Switch
: You do not want to change it.

2 Loading and Context Selection

2.1 Loading Redlog

At the beginning of each session REDLOG has to be loaded explicitly. This is done by inputing the command `load_package redlog;` from within a REDUCE session.

2.2 Context Selection

Fixing a context reflects the mathematical fact that first-order formulas are defined over fixed *languages* specifying, e.g., valid *function symbols* and *relation symbols* (*predicates*). After selecting a language, fixing a *theory* such as "the theory of ordered fields", allows to assign a semantics to the formulas. Both language and theory make up a REDLOG context. In addition, a context determines the internal representation of terms.

As first-order formulas are not defined unless the language is known, and meaningless unless the theory is known, it is impossible to enter a first-order formula into REDLOG without specifying a context:

```
REDUCE 3.6, 15-Jul-95, patched to 30 Aug 98 ...

1: load_package redlog;

2: f := a=0 and b=0;

***** select a context
```

See Section 1.1 [Contexts], page 2 for a summary of the available contexts OFSF, DVFSF, and ACFSF. A context can be selected by the `rlset` command:

rlset [*context* [*arguments...*]] Function
rlset *argument-list* Function

Set current context. Valid choices for *context* are OFSF (ordered fields standard form), DVFSF (discretely valued fields standard form), and ACFSF (algebraically closed fields standard form). With OFSF and ACFSF, there are no further arguments. With DVFSF an optional *dvf_class_specification* can be passed, which defaults to `0`. `rlset` returns the old setting as a list that can be saved to be passed to `rlset` later. When called with no arguments (or the empty list), `rlset` returns the current setting.

dvf_class_specification Data Structure

Zero, or a possibly negative prime q.

For $q = 0$ all computations are uniformly correct for all p-adic valuations. Both input and output then possibly involve a symbolic constant "p", which is being reserved.

For positive q, all computations take place wrt. the corresponding q-adic valuation.

For negative q, the "$-$" can be read as "up to", i.e., all computations are performed in such a way that they are correct for all p-adic valuations with $p \leq |q|$. In this case, the knowledge of an upper bound for p supports the quantifier elimination `rlqe`/`rlqea` [Stu98]. See Section 6.1 [Quantifier Elimination], page 27.

The user will probably have a "favorite" context reflecting their particular field of interest. To save the explicit declaration of the context with each session, REDLOG provides a global variable `rldeflang`, which contains a default context. This variable can be set already *before* loading '`redlog`'. This is typically done within the '`.reducerc`' profile:

```
lisp (rldeflang!* := '(ofsf));
```

Notice that the Lisp list representation has to be used here.

rldeflang!* Fluid

Default language. This can be bound to a default context before loading '`redlog`'. More precisely, `rldeflang!*` contains the arguments of `rlset` as a Lisp list. If `rldeflang!*` is non-nil, `rlset` is automatically executed on `rldeflang!*` when loading '`redlog`'.

In addition, REDLOG evaluates an environment variable `RLDEFLANG`. This allows to fix a default context within the shell already before starting REDUCE. The syntax for setting environment variables depends on the shell. For instance, in the GNU Bash or in the csh shell one would say `export RLDEFLANG=ofsf` or `setenv RLDEFLANG ofsf`, respectively.

RLDEFLANG Environment Variable

Default language. This may be bound to a context in the sense of the first argument of `rlset`. With `RLDEFLANG` set, any `rldeflang!*` binding is overloaded.

3 Format and Handling of Formulas

After loading REDLOG and selecting a context (see Chapter 2 [Loading and Context Selection], page 5), there are *first-order formulas* available as an additional type of symbolic expressions. That is, formulas are now subject to symbolic manipulation in the same way as, say, polynomials or matrices in conventional systems. There is nothing changed in the behavior of the builtin facilities and of other packages.

3.1 First-order Operators

Though the operators `and`, `or`, and `not` are already sufficient for representing boolean formulas, REDLOG provides a variety of other boolean operators for the convenient mnemonic input of boolean formulas.

not — Unary Operator
and — n-ary Infix Operator
or — n-ary Infix Operator
impl — Binary Infix Operator
repl — Binary Infix Operator
equiv — Binary Infix Operator

The infix operator precedence is from strongest to weakest: `and`, `or`, `impl`, `repl`, `equiv`.

See Section 3.6 [Extended Built-in Commands], page 10 for the description of extended `for`-loop actions that allow to comfortably input large systematic conjunctions and disjunctions.

REDUCE expects the user to know about the precedence of `and` over `or`. In analogy to `+` and `*`, there are thus no parentheses output around conjunctions within disjunctions. The following switch causes such subformulas to be bracketed anyway. See Section 1.3 [Conventions], page 4 for the notion of a "fix switch".

rlbrop — Fix Switch

Bracket all operators. By default this switch is on, which causes some private printing routines to be called for formulas: All subformulas are bracketed completely making the output more readable.

Besides the boolean operators introduced above, first-order logic includes the well-known *existential quantifiers* and *universal quantifiers* "$\exists$" and "$\forall$".

ex — Binary Operator
all — Binary Operator

These are the *quantifiers*. The first argument is the quantified variable, the second one is the matrix formula. Optionally, one can input a list of variables as first argument. This list is expanded into several nested quantifiers.

See Section 3.2 [Closures], page 8 for automatically quantifying all variables except for an exclusion list.

For convenience, we also have boolean constants for the truth values.

true Variable
false Variable

These algebraic mode variables are reserved. They serve as *truth values*.

3.2 Closures

rlall *formula* [*exceptionlist*] Function

Universal closure. *exceptionlist* is a list of variables empty by default. Returns *formula* with all free variables universally quantified, except for those in *exceptionlist*.

rlex *formula* [*exceptionlist*] Function

Existential closure. *exceptionlist* is a list of variables empty by default. Returns *formula* with all free variables existentially quantified, except for those in *exceptionlist*.

3.3 OFSF Operators

The OFSF context implements *ordered fields* over the language of *ordered rings*. Proceeding this way is very common in model theory since one wishes to avoid functions which are only partially defined, such as division in the language of ordered fields. Note that the OFSF quantifier elimination procedures (see Chapter 6 [Quantifier Elimination and Variants], page 27) for non-linear formulas actually operate over *real closed fields*. See Section 1.1 [Contexts], page 2 and Section 2.2 [Context Selection], page 5 for details on contexts.

equal Binary Infix operator
neq Binary Infix operator
leq Binary Infix operator
geq Binary Infix operator
lessp Binary Infix operator
greaterp Binary Infix operator

The above operators may also be written as `=`, `<>`, `<=`, `>=`, `<`, and `>`, respectively. For OFSF there is specified that all right hand sides must be zero. Non-zero right hand sides in the input are hence subtracted immediately to the corresponding left hand sides. There is a facility to input *chains* of the above relations, which are also expanded immediately:

```
a<>b<c>d=f
     ⇒ a-b <> 0 and b-c < 0 and c-d > 0 and d-f = 0
```

Here, only adjacent terms are related to each other.

Though we use the language of ordered rings, the input of integer reciprocals is allowed and treated correctly interpreting them as constants for rational numbers. There are two switches that allow to input arbitrary reciprocals, which are then resolved into proper formulas in various reasonable ways. The user is welcome to experiment with switches like the following, which are not marked as *fix switches*. See Section 1.3 [Conventions], page 4 for the classification of REDLOG switches.

rlnzden Switch
rlposden Switch

Non-zero/positive denominators. Both switches are off by default. If both `rlnzden` and `rlposden` are on, the latter is active. Activating one of them, allows the input of reciprocal terms. With `rlnzden` on, these terms are assumed to be non-zero, and resolved by multiplication. When occurring with ordering relations the reciprocals are resolved by multiplication with their square preserving the sign.

```
(a/b)+c=0 and (a/d)+c>0;
                                       2
     ⇒ a + b*c = 0 and a*d + c*d  > 0
```

Turning `rlposden` on, guarantees the reciprocals to be strictly positive which allows simple, i.e. non-square, multiplication also with ordering relations.

```
(a/b)+c=0 and (a/d)+c>0;
     ⇒ a + b*c = 0 and a + c*d > 0
```

The non-zeroness or positivity assumptions made by using the above switches can be stored in a variable, and then later be passed as a *theory* (see Section 4.1 [Standard Simplifier], page 14) to certain REDLOG procedures. Optionally, the system can be forced to add them to the formula at the input stage:

rladdcond Switch

Add condition. This is off by default. With `rladdcond` on, non-zeroness and positivity assumptions made due to the switches `rlnzden` and `rlposden` are added to the formula at the input stage. With `rladdcond` and `rlposden` on we get for instance:

```
(a/b)+c=0 and (a/d)+c>0;
     ⇒ (b <> 0 and a + b*c = 0) and (d > 0 and a + c*d > 0)
```

3.4 DVFSF Operators

Discretely valued fields are implemented as a one-sorted language using the operators |, ||, ~, and /~, which encode $\leq$, $<$, $=$, and $\neq$ in the value group, respectively. For details see [Wei88], [Stu98], or [DS99].

equal Binary Infix operator
neq Binary Infix operator
div Binary Infix operator
sdiv Binary Infix operator
assoc Binary Infix operator
nassoc Binary Infix operator

The above operators may also be written as =, <>, |, ||, ~, and /~, respectively. Integer reciprocals in the input are resolved correctly. DVFSF allows the input of *chains* in analogy to OFSF. See Section 3.3 [OFSF Operators], page 8 for details.

With the DVFSF operators there is no treatment of parametric denominators available.

3.5 ACFSF Operators

equal Binary Infix operator
neq Binary Infix operator

The above operators may also be written as =, <>. As for OFSF, it is specified that all right hand sides must be zero. In analogy to OFSF, ACFSF allows also the input of *chains* and an appropriate treatment of parametric denominators in the input. See Section 3.3 [OFSF Operators], page 8 for details.

Note that the switch `rlposden` (see Section 3.3 [OFSF Operators], page 8) makes no sense for algebraically closed fields.

3.6 Extended Built-in Commands

Systematic conjunctions and disjunctions can be constructed in the algebraic mode in analogy to, e.g., `for ... sum ...`:

mkand for loop action
mkor for loop action

Make and/or. Actions for the construction of large systematic conjunctions/disjunctions via for loops.

```
for i:=1:3 mkand
   for j:=1:3 mkor
      if j<>i then mkid(x,i)+mkid(x,j)=0;
```

```
⇒ true and (false or false or x1 + x2 = 0
            or x1 + x3 = 0)
    and (false or x1 + x2 = 0 or false
         or x2 + x3 = 0)
    and (false or x1 + x3 = 0 or x2 + x3 = 0
         or false)
```

Here the truth values come into existence due to the internal implementation of `for`-loops. They are always neutral in their context, and can be easily removed via simplification (see Section 4.1 [Standard Simplifier], page 14, see Section 3.7 [Global Switches], page 11).

The REDUCE substitution command `sub` can be applied to formulas using the usual syntax.

substitution_list Data Structure

substitution_list is a list of equations each with a kernel left hand side.

sub *substitution_list formula* Function

Substitute. Returns the formula obtained from *formula* by applying the substitutions given by *substitution_list*.

```
sub(a=x,ex(x,x-a<0 and all(x,x-b>0 or ex(a,a-b<0))));
    ⇒ ex x0 ( - x + x0 < 0 and all x0 (
            - b + x0 > 0 or ex a (a - b < 0)))
```

`sub` works in such a way that equivalent formulas remain equivalent after substitution. In particular, quantifiers are treated correctly.

part *formula n1* [*n2* [*n3*...]] Function

Extract a part. The `part` of *formula* is implemented analogously to that for built-in types: in particular the 0th part is the operator.

Compare `rlmatrix` (see Section 3.8 [Basic Functions on Formulas], page 12) for extracting the *matrix part* of a formula, i.e., removing *all* initial quantifiers.

length *formula* Function

Length of *formula*. This is the number of arguments to the top-level operator. The length is of particular interest with the n-ary operators `and` and `or`. Notice that `part(`*formula*`,length(`*formula*`))` is the part of largest index.

3.7 Global Switches

There are three global switches that do not belong to certain procedures, but control the general behavior of REDLOG.

rlsimpl Switch

Simplify. By default this switch is off. With this switch on, the function `rlsimpl` is applied at the expression evaluation stage. See Section 4.1 [Standard Simplifier], page 14.

Automatically performing formula simplification at the evaluation stage is very similar to the treatment of polynomials or rational functions, which are converted to some normal form. For formulas, however, the simplified equivalent is by no means canonical.

rlrealtime Switch

Real time. By default this switch is off. If on it protocols the wall clock time needed for REDLOG commands in seconds. In contrast to the built-in `time` switch, the time is printed *above* the result.

rlverbose Advanced Switch

Verbose. By default this switch is off. It toggles verbosity output with some REDLOG procedures. The verbosity output itself is not documented.

3.8 Basic Functions on Formulas

rlatnum *formula* Function

Number of atomic formulas. Returns the number of atomic formulas contained in *formula*. Mind that truth values are not considered atomic formulas.

multiplicity_list Data Structure

A list of 2-element-lists containing an object and the number of its occurrences. Names of functions returning *multiplicity_lists* typically end on "ml".

rlatl *formula* Function

List of atomic formulas. Returns the set of atomic formulas contained in *formula* as a list.

rlatml *formula* Function

Multiplicity list of atomic formulas. Returns the atomic formulas contained in *formula* in a *multiplicity_list*.

rlifacl *formula* Function

List of irreducible factors. Returns the set of all irreducible factors of the nonzero terms occurring in *formula*.

```
rlifacl(x**2-1=0);
    ⇒ {x + 1,x - 1}
```

rlifacml *formula* Function

Multiplicity list of irreducible factors. Returns the set of all irreducible factors of the nonzero terms occurring in *formula* in a *multiplicity_list*.

rlterml *formula* Function

List of terms. Returns the set of all nonzero terms occurring in *formula*.

rltermml *formula* Function

Multiplicity list of terms. Returns the set of all nonzero terms occurring in *formula* in a *multiplicity_list*.

rlvarl *formula* Function

Variable lists. Returns both the list of variables occurring freely and that of the variables occurring boundly in *formula* in a two-element list. Notice that the two member lists are not necessarily disjoint.

rlfvarl *formula* Function

Free variable list. Returns the variables occurring freely in *formula* as a list.

rlbvarl *formula* Function

Bound variable list. Returns the variables occurring boundly in *formula* as a list.

rlstruct *formula* [*kernel*] Function

Structure of a formula. *kernel* is `v` by default. Returns a list `{f,sl}`. `f` is constructed from *formula* by replacing each occurrence of a term with a kernel constructed by concatenating a number to *kernel*. The *substitution_list* `sl` contains all substitutions to obtain *formula* from `f`.

```
rlstruct(x*y=0 and (x=0 or y>0),v);
     ⇒ {v1 = 0 and (v2 = 0 or v3 > 0),
        {v1 = x*y,v2 = x,v3 = y}}
```

rlifstruct *formula* [*kernel*] Function

Irreducible factor structure of a formula. *kernel* is `v` by default. Returns a list `{f,sl}`. `f` is constructed from *formula* by replacing each occurrence of an irreducible factor with a kernel constructed by adding a number to *kernel*. The returned *substitution_list* `sl` contains all substitutions to obtain *formula* from `f`.

```
rlstruct(x*y=0 and (x=0 or y>0),v);
     ⇒ {v1*v2 = 0 and (v1 = 0 or v2 > 0),
        {v1 = x,v2 = y}}
```

rlmatrix *formula* Function

Matrix computation. Drops all *leading* quantifiers from *formula*.

4 Simplification

The goal of simplifying a first-order formula is to obtain an equivalent formula over the same language that is somehow simpler. REDLOG knows three kinds of simplifiers that focus mainly on reducing the size of the given formula: The standard simplifier, tableau simplifiers, and Groebner simplifiers. The OFSF versions of these are described in [DS97].

The ACFSF versions are the same as the OFSF versions except for techniques that are particular to ordered fields such as treatment of square sums in connection with ordering relations.

For DVFSF there is no Groebner simplifier available. The parts of the standard simplifier that are particular to valued fields are described in [DS99]. The tableau simplification is straightforwardly derived from the *smart simplifications* described there.

Besides reducing the size of formulas, it is a reasonable simplification goal, to reduce the degree of the quantified variables. Our method of decreasing the degree of quantified variables is described for OFSF in [DSW98]. A suitable variant is available also in ACFSF but not in DVFSF.

4.1 Standard Simplifier

The *Standard Simplifier* is a fast simplification algorithm that is frequently called internally by other REDLOG algorithms. It can be applied automatically at the expression evaluation stage by turning on the switch `rlsimpl` (see Section 3.7 [Global Switches], page 11).

theory Data Structure

A list of atomic formulas assumed to hold.

rlsimpl *formula* [*theory*] Function

Simplify. *formula* is simplified recursively such that the result is equivalent under the assumption that *theory* holds. Default for *theory* is the empty theory `{}`. Theory inconsistency may but need not be detected by `rlsimpl`. If *theory* is detected to be inconsistent, a corresponding error is raised. Note that under an inconsistent theory, *any* formula is equivalent to the input, i.e., the result is meaningless. *theory* should thus be chosen carefully.

4.1.1 General Features of the Standard Simplifier

The standard simplifier `rlsimpl` includes the following features common to all contexts:

- Replacement of atomic subformulas by simpler equivalents. These equivalents are not necessarily atomic (switches `rlsiexpl`, `rlsiexpla`, see Section 4.1.2 [General Standard Simplifier Switches], page 15). For details on the simplification on the atomic formula level, see Section 4.1.3 [OFSF-specific Simplifications], page 17, Section 4.1.5 [ACFSF-specific Simplifications], page 19, and Section 4.1.7 [DVFSF-specific Simplifications], page 19.

- Proper treatment of truth values.
- Flattening nested n-ary operator levels and resolving involutive applications of `not`.
- Dropping `not` operator with atomic formula arguments by changing the relation of the atomic formula appropriately. The languages of all contexts allow to do so.
- Changing `repl` to `impl`.
- Producing a canonical ordering among the atomic formulas on a given level (switch `rlsisort`, see Section 4.1.2 [General Standard Simplifier Switches], page 15).
- Recognizing equal subformulas on a given level (switch `rlsichk`, see Section 4.1.2 [General Standard Simplifier Switches], page 15).
- Passing down information that is collected during recursion (switches `rlsism`, `rlsiidem`, see Section 4.1.2 [General Standard Simplifier Switches], page 15). The technique of *implicit theories* used for this is described in detail in [DS97] for OFSF/ACFSF, and in [DS99] for DVFSF.
- Considering interaction of atomic formulas on the same level and interaction with information inherited from higher levels (switch `rlsism`, see Section 4.1.2 [General Standard Simplifier Switches], page 15). The *smart simplification* techniques used for this are beyond the scope of this manual. They are described in detail in [DS97] for OFSF/ACFSF, and in [DS99] for DVFSF.

4.1.2 General Standard Simplifier Switches

rlsiexpla Switch

Simplify explode always. By default this switch is on. It is relevant with simplifications that allow to split one atomic formula into several simpler ones. Consider, e.g., the following simplification toggled by the switch `rlsipd` (see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17). With `rlsiexpla` on, we obtain:

```
f := (a - 1)**3 * (a + 1)**4 >=0;
         7    6      5      4      3      2
     ⇒ a  + a  - 3*a  - 3*a  + 3*a  + 3*a  - a - 1 >= 0

rlsimpl f;
```

```
        ⇒ a - 1 >= 0 or a + 1 = 0
```

With `rlsiexpla` off, `f` will simplify as in the description of the switch `rlsipd`. `rlsiexpla` is not used in the DVFSF context. The DVFSF simplifier behaves like `rlsiexpla` on.

rlsiexpl Switch

Simplify explode. By default this switch is on. Its role is very similar to that of `rlsiexpla`, but it considers the operator the scope of which the atomic formula occurs in: With `rlsiexpl` on

```
   7    6      5      4      3      2
a  + a  - 3*a  - 3*a  + 3*a  + 3*a  - a - 1 >= 0
```

simplifies as in the description of the switch `rlsiexpla` whenever it occurs in a disjunction, and it simplifies as in the description of the switch `rlsipd` (see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17) else. `rlsiexpl` is not used in the DVFSF context. The DVFSF simplifier behaves like `rlsiexpla` on.

The user is not supposed to alter the settings of the following *fix switches* (see Section 1.3 [Conventions], page 4):

rlsism Fix Switch

Simplify smart. By default this switch is on. See the description of the function `rlsimpl` (see Section 4.1 [Standard Simplifier], page 14) for its effects.

```
rlsimpl(x>0 and x+1<0);
     ⇒ false
```

rlsichk Fix Switch

Simplify check. By default this switch is on enabling checking for equal sibling subformulas:

```
rlsimpl((x>0 and x-1<0) or (x>0 and x-1<0));
     ⇒ (x>0 and x-1<0)
```

rlsiidem Fix Switch

Simplify idempotent. By default this switch is on. It is relevant only with switch `rlsism` on. Its effect is that `rlsimpl` (see Section 4.1 [Standard Simplifier], page 14) is idempotent in the very most cases, i.e., an application of `rlsimpl` to an already simplified formula yields the formula itself.

rlsiso Fix Switch

Simplify sort. By default this switch is on. It toggles the sorting of the atomic formulas on the single levels.

```
rlsimpl((a=0 and b=0) or (b=0 and a=0));
     ⇒ a = 0 and b = 0
```

4.1.3 OFSF-specific Simplifications

In the OFSF context, the atomic formula simplification includes the following:

- Evaluation of variable-free atomic formulas to truth values.
- Make the left hand sides primitive over the integers with positive head coefficient.
- Evaluation of trivial square sums to truth values (switch `rlsisqf`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17). Additive splitting of trivial square sums (switch `rlsitsqspl`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17).
- In ordering inequalities, perform parity decomposition (similar to squarefree decomposition) of terms (switch `rlsipd`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17) with the option to split an atomic formula multiplicatively into two simpler ones (switches `rlsiexpl` and `rlsiexpla`, see Section 4.1.2 [General Standard Simplifier Switches], page 15).
- In equations and non-ordering inequalities, replace left hand sides by their squarefree parts (switch `rlsiatdv`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17). Optionally, perform factorization of equations and inequalities (switch `rlsifac`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17, switches `rlsiexpl` and `rlsiexpla`, see Section 4.1.2 [General Standard Simplifier Switches], page 15).

For further details on the simplification in ordered fields see the article [DS97].

4.1.4 OFSF-specific Standard Simplifier Switches

rlsipw Switch

Simplification prefer weak orderings. Prefers weak orderings in contrast to strict orderings with implicit theory simplification. `rlsipw` is off by default, which leads to the following behavior:

```
rlsimpl(a<>0 and (a>=0 or b=0));
     ⇒ a <> 0 and (a > 0 or b = 0)
```

This meets the simplification goal of small satisfaction sets for the atomic formulas. Turning on `rlsipw` will instead yield the following:

```
rlsimpl(a<>0 and (a>0 or b=0));
     ⇒ a <> 0 and (a >= 0 or b = 0)
```

Here we meet the simplification goal of convenient relations when strict orderings are considered inconvenient.

rlsipo Switch

Simplification prefer orderings. Prefers orderings in contrast to inequalities with implicit theory simplification. `rlsipo` is on by default, which leads to the following behavior:

```
rlsimpl(a>=0 and (a<>0 or b=0));
     ⇒ a >= 0 and (a > 0 or b = 0)
```

This meets the simplification goal of small satisfaction sets for the atomic formulas. Turning it on leads, e.g., to the following behavior:

```
rlsimpl(a>=0 and (a>0 or b=0));
     ⇒ a >= 0 and (a <> 0 or b = 0)
```

Here, we meet the simplification goal of convenient relations when orderings are considered inconvenient.

rlsiatadv Switch

Simplify atomic formulas advanced. By default this switch is on. Enables sophisticated atomic formula simplifications based on squarefree part computations and recognition of trivial square sums.

```
rlsimpl(a**2 + 2*a*b + b**2 <> 0);
     ⇒ a+b <> 0

rlsimpl(a**2 + b**2 + 1 > 0);
     ⇒ true
```

Furthermore, splitting of trivial square sums (switch `rlsitsqspl`), parity decompositions (switch `rlsipd`), and factorization of equations and inequalities (switch `rlsifac`) are enabled.

rlsitsqspl Switch

Simplify split trivial square sum. This is on by default. It is ignored with `rlsiadv` off. Trivial square sums are split additively depending on `rlsiexpl` and `rlsiexpla` (see Section 4.1.2 [General Standard Simplifier Switches], page 15):

```
rlsimpl(a**2+b**2>0);
     ⇒ a <> 0 or b <> 0
```

rlsipd Switch

Simplify parity decomposition. By default this switch is on. It is ignored with `rlsiatadv` off. `rlsipd` toggles the parity decomposition of terms occurring with ordering relations.

```
f := (a - 1)**3 * (a + 1)**4 >= 0;
          7    6      5      4      3      2
     ⇒ a  + a  - 3*a  - 3*a  + 3*a  + 3*a  - a - 1 >= 0

rlsimpl f;
          3    2
     ⇒ a  + a  - a - 1 >= 0
```

The atomic formula is possibly split into two parts according to the setting of the switches `rlsiexpl` and `rlsiexpla` (see Section 4.1.2 [General Standard Simplifier Switches], page 15).

rlsifac Switch

Simplify factorization. By default this switch is on. It is ignored with `rlsiatadv` off. Splits equations and inequalities via factorization of their left hand side terms into a disjunction or a conjunction, respectively. This is done in dependence on `rlsiexpl` and `rlsiexpla` (see Section 4.1.2 [General Standard Simplifier Switches], page 15).

4.1.5 ACFSF-specific Simplifications

In the ACFSF case the atomic formula simplification includes the following:

- Evaluation of variable-free atomic formulas to truth values.
- Make the left hand sides primitive over the integers with positive head coefficient.
- Replace left hand sides of atomic formulas by their squarefree parts (switch `rlsiatdv`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17). Optionally, perform factorization of equations and inequalities (switch `rlsifac`, see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17, switches `rlsiexpl` and `rlsiexpla`, see Section 4.1.2 [General Standard Simplifier Switches], page 15).

For details see the description of the simplification for ordered fields in [DS97]. This can be easily adapted to algebraically closed fields.

4.1.6 ACFSF-specific Standard Simplifier Switches

The switches `rlsiatadv` and `rlsifac` have the same effects as in the OFSF context (see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17).

4.1.7 DVFSF-specific Simplifications

In the DVFSF case the atomic formula simplification includes the following:

- Evaluation of variable-free atomic formulas to truth values provided that p is known.
- Equations and inequalities can be treated as in ACFSF (see Section 4.1.5 [ACFSF-specific Simplifications], page 19). Moreover powers of p can be cancelled.
- With valuation relations, the GCD of both sides is cancelled and added appropriately as an equation or inequality.

- Valuation relations involving zero sides can be evaluated or at least turned into equations or inequalities.
- For concrete p, integer coefficients with valuation relations can be replaced by a power of p on one side of the relation.
- For unspecified p, polynomials in p can often be replaced by one monomial.
- For unspecified p, valuation relations containing a monomial in p on one side, and an integer on the other side can be transformed into `z ~ 1` or `z /~ 1`, where `z` is an integer.

For details on simplification in p-adic fields see the article [DS99].

Atomic formulas of the form `z ~ 1` or `z /~ 1`, where `z` is an integer, can be split into several ones via integer factorization. This simplification is often reasonable on final results. It explicitly discovers those primes p for which the formula holds. There is a special function for this simplification:

rlexplats *formula* Function

Explode atomic formulas. Factorize atomic formulas of the form `z ~ 1` or `z /~ 1`, where `z` is an integer. `rlexplats` obeys the switches `rlsiexpla` and `rlsiexpl` (see Section 4.1.2 [General Standard Simplifier Switches], page 15), but not `rlsifac` (see Section 4.1.8 [DVFSF-specific Standard Simplifier Switches], page 20).

4.1.8 DVFSF-specific Standard Simplifier Switches

The context DVFSF knows no special simplifier switches, and ignores the general switches `rlsiexpla` and `rlsiexpl` (see Section 4.1.2 [General Standard Simplifier Switches], page 15). It behaves like `rlsiexpla` on. The simplifier contains numerous sophisticated simplifications for atomic formulas in the style of `rlsiatadv` on (see Section 4.1.4 [OFSF-specific Standard Simplifier Switches], page 17).

rlsifac Switch

Simplify factorization. By default this switch is on. Toggles certain simplifications that require *integer* factorization. See Section 4.1.7 [DVFSF-specific Simplifications], page 19 for details.

4.2 Tableau Simplifier

Although our standard simplifier (see Section 4.1 [Standard Simplifier], page 14) already combines information located on different boolean levels, it preserves the basic boolean structure of the formula. The tableau methods, in contrast, provide a technique for changing the boolean structure of a formula by constructing case distinctions. Compared to the standard simplifier they are much slower. For details on tableau simplification see [DS97].

cdl Data Structure

Case distinction list. This is a list of atomic formulas considered as a disjunction.

rltab *formula cdl* Function

Tableau method. The result is a tableau wrt. *cdl*, i.e., a simplified equivalent of the disjunction over the specializations wrt. all atomic formulas in *cdl*.

```
rltab((a = 0 and (b = 0 or (d = 0 and e = 0))) or
   (a = 0 and c = 0),{a=0,a<>0});
     ⇒ (a = 0 and (b = 0 or c = 0 or (d = 0 and e = 0)))
```

rlatab *formula* Function

Automatic tableau method. Tableau steps wrt. a case distinction over the signs of all terms occurring in *formula* are computed and the best result, i.e., the result with the minimal number of atomic formulas is returned.

rlitab *formula* Function

Iterative automatic tableau method. *formula* is simplified by iterative applications of `rlatab`. The exact procedure depends on the switch `rltabib`.

rltabib Switch

Tableau iterate branch-wise. By default this switch is on. It controls the procedure `rlitab`. If `rltabib` is off, `rlatab` is iteratively applied to the argument formula as long as shorter results can be obtained. In case `rltabib` is on, the corresponding next tableau step is not applied to the last tableau result but independently to each single branch. The iteration stops when the obtained formula is not smaller than the corresponding input.

4.3 Groebner Simplifier

The Groebner simplifier is not available in the DVFSF context. It considers order theoretical and algebraic simplification rules between the atomic formulas of the input formula. Currently the Groebner simplifier is not idempotent. The name is derived from the main mathematical tool used for simplification: Computing Groebner bases of certain subsets of terms occurring in the atomic formulas.

For calling the Groebner simplifier there are the following functions:

rlgsc *formula* [*theory*] Function
rlgsd *formula* [*theory*] Function
rlgsn *formula* [*theory*] Function

Groebner simplifier. *formula* is a quantifier-free formula. Default for *theory* is the empty theory `{}`. The functions differ in the boolean normal form that is computed at the beginning. `rlgsc` computes a conjunctive normal form, `rlgsd` computes a disjunctive normal form, and `rlgsn` heuristically decides for either a conjunctive or a disjunctive normal form depending on the structure of *formula*. After computing the corresponding normal form, the formula is simplified using Groebner simplification techniques. The returned formula is equivalent to the input formula wrt. *theory*.

```
rlgsd(x=0 and ((y = 0 and x**2+2*y > 0) or
        (z=0 and x**3+z >= 0)));
    ⇒ x = 0 and z = 0

rlgsc(x neq 0 or ((y neq 0 or x**2+2*x*y <= 0) and
        (z neq 0 or x**3+z < 0)));
     ⇒ x <> 0 or z <> 0
```

The heuristic used by `rlgsn` is intended to find the smaller boolean normal form among CNF an DNF. Note that, anyway, the simplification of the smaller normal form can lead to a larger final result than that of the larger one.

The Groebner simplifiers use the Groebner package of REDUCE to compute the various Groebner bases. By default, the `revgradlex` term order is used, and no optimizations of the order between the variables are applied. The other switches and variables of the Groebner package are not controlled by the Groebner simplifier. They can be adjusted by the user.

In contrast to the standard simplifier `rlsimpl` (see Section 4.1 [Standard Simplifier], page 14), the Groebner simplifiers can in general produce formulas containing more atomic formulas than the input. This cannot happen if the switches `rlgsprod`, `rlgsred`, and `rlgssub` are off and the input formula is a simplified boolean normal form.

The functionality of the Groebner simplifiers `rlgsc`, `rlgsd`, and `rlgsn` is controlled by numerous switches. In most cases the default settings lead to a good simplification.

rlgsrad Switch

Groebner simplifier radical membership test. By default this switch is on. If the switch is on the Groebner simplifier does not only use ideal membership tests for simplification but also radical membership tests. This leads to better simplifications but takes considerably more time.

rlgssub Switch

Groebner simplifier substitute. By default this switch is on. Certain subsets of atomic formulas are substituted by equivalent ones. Both the number of atomic formulas and the complexity of the terms may increase or decrease independently.

rlgsbnf Switch

Groebner simplifier boolean normal form. By default this switch is on. Then the simplification starts with a boolean normal form computation. If the switch is off, the simplifiers expect a boolean normal form as the argument *formula*.

rlgsred Switch

Groebner simplifier reduce polynomials. By default this switch is on. It controls the reduction of the terms wrt. the computed Groebner bases. The number of atomic formulas is never increased. Mind that by reduction the terms can become more complicated.

rlgsvb Advanced Switch

Groebner simplifier verbose. By default this switch is on. It toggles verbosity output of the Groebner simplifier. Verbosity output is given if and only if both `rlverbose` (see Section 3.7 [Global Switches], page 11) and `rlgsvb` are on.

rlgsprod Advanced Switch

Groebner simplifier product. By default this switch is off. If this switch is on then conjunctions of inequalities and disjunctions of equations are contracted multiplicatively to one atomic formula. This reduces the number of atomic formulas but in most cases it raises the complexity of the terms. Most simplifications recognized by considering products are detected also with `rlgsprod` off.

rlgserf Advanced Switch

Groebner simplifier evaluate reduced form. By default this switch is on. It controls the evaluation of the atomic formulas to truth values. If this switch is on, the standard simplifier (see Section 4.1 [Standard Simplifier], page 14) is used to evaluate atomic formulas. Otherwise atomic formulas are evaluated only if their left hand side is a domain element.

rlgsutord Advanced Switch

Groebner simplifier user defined term order. By default this switch is off. Then all Groebner basis computations and reductions are performed with respect to the `revgradlex` term order. If this switch is on, the Groebner simplifier minds the term order selected with the `torder` statement. For passing a variable list to `torder`, note that `rlgsradmemv!*` is used as the tag variable for radical membership tests.

rlgsradmemv!* Fluid

Radical membership test variable. This fluid contains the tag variable used for the radical membership test with switch `rlgsrad` on. It can be used to pass the variable explicitly to `torder` with switch `rlgsutord` on.

4.4 Degree Decreaser

The quantifier elimination procedures of REDLOG (see Section 6.1 [Quantifier Elimination], page 27) obey certain degree restrictions on the bound variables. For this reason, there are degree-decreasing simplifiers available, which are automatically applied by the corresponding quantifier elimination procedures. There is no degree decreaser for the DVFSF context available.

rldecdeg *formula* Function

Decrease degrees. Returns a formula equivalent to *formula*, hopefully decreasing the degrees of the bound variables. In the OFSF context there are in general some sign conditions on the variables added, which slightly increases the number of contained atomic formulas.

```
rldecdeg ex({b,x},m*x**4711+b**8>0);
    ⇒ ex b (b >= 0 and ex x (b + m*x > 0))
```

rldecdeg1 *formula* [*varlist*] Function

Decrease degrees subroutine. This provides a low-level entry point to the degree decreaser. The variables to be decreased are not determined by regarding quantifiers but are explicitly given by *varlist*, where the empty *varlist* selects *all* free variables of `f`. The return value is a list `{f,l}`. `f` is a formula, and `l` is a list `{...,{v,d},...}`, where `v` is from *varlist* and `d` is an integer. `f` has been obtained from *formula* by substituting `v` for `v**d`. The sign conditions necessary with the OFSF context are not generated automatically, but have to be constructed by hand for the variables `v` with even degree `d` in `l`.

```
rldecdeg1(m*x**4711+b**8>0,{b,x});
    ⇒ {b + m*x > 0,{{x,4711},{b,8}}}
```

5 Normal Forms

5.1 Boolean Normal Forms

For computing small boolean normal forms, see also the documentation of the procedures `rlgsc` and `rlgsd` (Section 4.3 [Groebner Simplifier], page 21).

rlcnf *formula* — Function

Conjunctive normal form. *formula* is a quantifier-free formula. Returns a conjunctive normal form of *formula*.

```
rlcnf(a = 0 and b = 0 or b = 0 and c = 0);
    ⇒ (a = 0 or c = 0) and b = 0
```

rldnf *formula* — Function

Disjunctive normal form. *formula* is a quantifier-free formula. Returns a disjunctive normal form of *formula*.

```
rldnf((a = 0 or b = 0) and (b = 0 or c = 0));
    ⇒ (a = 0 and c = 0) or b = 0
```

rlbnfsm — Switch

Boolean normal form smart. By default this switch is off. If on, *simplifier recognized implication* [DS97] is applied by `rlcnf` and `rldnf`. This leads to smaller normal forms but is considerably time consuming.

rlbnfsac — Fix Switch

Boolean normal forms subsumption and cut. By default this switch is on. With boolean normal form computation, subsumption and cut strategies are applied by `rlcnf` and `rldnf` to decrease the number of clauses. We give an example:

```
rldnf(x=0 and y<0 or x=0 and y>0 or x=0 and y<>0 and z=0);
    ⇒ (x = 0 and y <> 0)
```

5.2 Miscellaneous Normal Forms

rlnnf *formula* — Function

Negation normal form. Returns a *negation normal form* of *formula*. This is an `and`-`or`-combination of atomic formulas. Note that in all contexts, we use languages such that all negations can be encoded by relations (see Chapter 3 [Format and Handling of Formulas], page 7). We give an example:

```
rlnnf(a = 0 equiv b > 0);
    ⇒ (a = 0 and b > 0) or (a <> 0 and b <= 0)
```

`rlnnf` accepts formulas containing quantifiers, but it does not eliminate quantifiers.

rlpnf *formula* Function

Prenex normal form. Returns a prenex normal form of *formula*. The number of quantifier changes in the result is minimal among all prenex normal forms that can be obtained from *formula* by only moving quantifiers to the outside.

When *formula* contains two quantifiers with the same variable such as in

$$\exists x(x = 0) \land \exists x(x \neq 0)$$

there occurs a name conflict. It is resolved according to the following rules:

- Every bound variable that stands in conflict with any other variable is renamed.
- Free variables are never renamed.

Hence `rlpnf` applied to the above example formula yields

```
rlpnf(ex(x,x=0) and ex(x,x<>0));
    ⇒ ex x0 ex x1 (x0 = 0 and x1 <> 0)
```

rlapnf *formula* Function

Anti-prenex normal form. Returns a formula equivalent to *formula* where all quantifiers are moved to the inside as far as possible.

```
rlapnf ex(x,all(y,x=0 or (y=0 and x=z)));
    ⇒ ex x (x = 0) or (all y (y = 0) and ex x (x - z = 0))
```

rltnf *formula terml* Function

Term normal form. *terml* is a list of terms. This combines DNF computation with tableau ideas (see Section 4.2 [Tableau Simplifier], page 20). A typical choice for *terml* is `rlterml` *formula*. If the switch `rltnft` is off, then `rltnf(`*formula*`,rlterml` *formula*`)` returns a DNF.

rltnft Switch

Term normal form tree variant. By default this switch is on causing `rltnf` to return a deeply nested formula.

6 Quantifier Elimination and Variants

Quantifier elimination computes quantifier-free equivalents for given first-order formulas. For OFSF and DVFSF we use a technique based on elimination set ideas [Wei88]. The OFSF implementation is restricted to at most quadratic occurrences of the quantified variables, but includes numerous heuristic strategies for coping with higher degrees. See [LW93], [Wei97] for details on the method. The DVFSF implementation is restricted to formulas that are linear in the quantified variables; the method is described in detail in [Stu98]. The ACFSF quantifier elimination is based on *comprehensive Groebner basis* computation; there are no degree restrictions for this context [Wei92].

6.1 Quantifier Elimination

rlqe *formula* [*theory*] Function

Quantifier elimination. Returns a quantifier-free equivalent of *formula* (wrt. *theory*). In the contexts OFSF and DVFSF, *formula* has to obey certain degree restrictions. There are various techniques for decreasing the degree of the input and of intermediate results built in. In case that not all variables can be eliminated, the resulting formula is not quantifier-free but still equivalent.

For degree decreasing heuristics see, e.g., Section 4.4 [Degree Decreaser], page 24, or the switches `rlqeqsc`/`rlqesqsc`.

elimination_answer Data Structure

A list of *condition–solution pairs*, i.e., a list of pairs consisting of a quantifier-free formula and a set of equations.

rlqea *formula* [*theory*] Function

Quantifier elimination with answer. Returns an *elimination_answer* obtained the following way: *formula* is wlog. prenex. All quantifier blocks but the outermost one are eliminated. For this outermost block, the constructive information obtained by the elimination is saved:

- In case the considered block is existential, for each evaluation of the free variables we know the following: Whenever a *condition* holds, then *formula* is `true` under the given evaluation, and the *solution* is *one* possible evaluation for the outer block variables satisfying the matrix.
- The universally quantified case is dual: Whenever a *condition* is false, then *formula* is `false`, and the *solution* is *one* possible counterexample.

As an example we show how to find conditions and solutions for a system of two linear constraints:

```
rlqea ex(x,x+b1>=0 and a2*x+b2<=0);
            2                                      - b2
    ⇒ {{a2 *b1 - a2*b2 >= 0 and a2 <> 0,{x = -------}},
                                                   a2
        {a2 < 0 or (a2 = 0 and b2 <= 0),{x = infinity1}}}
```

The answer can contain constants named `infinity` or `epsilon`, both indexed by a number: All `infinity`'s are positive and infinite, and all `epsilon`'s are positive and infinitesimal wrt. the considered field. Nothing is known about the ordering among the `infinity`'s and `epsilon`'s though this can be relevant for the points to be solutions. With the switch `rounded` on, the `epsilon`'s are evaluated to zero. `rlqea` is not available in the context ACFSF.

rlqeqsc — Switch
rlqesqsc — Switch

Quantifier elimination (super) quadratic special case. By default these switches are off. They are relevant only in OFSF. If turned on, alternative elimination sets are used for certain special cases by `rlqe`/`rlqea` and `rlgqe`/`rlgqea`. (see Section 6.2 [Generic Quantifier Elimination], page 29). They will possibly avoid violations of the degree restrictions, but lead to larger results in general. Former versions of REDLOG without these switches behaved as if `rlqeqsc` was on and `rlqesqsc` was off.

rlqedfs — Advanced Switch

Quantifier elimination depth first search. By default this switch is off. It is also ignored in the ACFSF context. It is ignored with the switch `rlqeheu` on, which is the default for OFSF. Turning `rlqedfs` on makes `rlqe`/`rlqea` and `rlgqe`/`rlgqea` (see Section 6.2 [Generic Quantifier Elimination], page 29) work in a depth first search manner instead of breadth first search. This saves space, and with decision problems, where variable-free atomic formulas can be evaluated to truth values, it might save time. In general, it leads to larger results.

rlqeheu — Advanced Switch

Quantifier elimination search heuristic. By default this switch is on in OFSF and off in DVFSF. It is ignored in ACFSF. Turning `rlqeheu` on causes the switch `rlqedfs` to be ignored. `rlqe`/`rlqea` and `rlgqe`/`rlgqea` (see Section 6.2 [Generic Quantifier Elimination], page 29) will then decide between breadth first search and depth first search for each quantifier block, where DFS is chosen when the problem is a decision problem.

rlqepnf Advanced Switch

Quantifier elimination compute prenex normal form. By default this switch is on, which causes that `rlpnf` (see Section 5.2 [Miscellaneous Normal Forms], page 25) is applied to *formula* before starting the elimination process. If the argument formula to `rlqe`/`rlqea` or `rlgqe`/`rlgqea` (see Section 6.2 [Generic Quantifier Elimination], page 29) is already prenex, this switch can be turned off. This may be useful with formulas containing `equiv` since `rlpnf` applies `rlnnf`, (see Section 5.2 [Miscellaneous Normal Forms], page 25), and resolving equivalences can double the size of a formula. `rlqepnf` is ignored in ACFSF, since NNF is necessary for elimination there.

rlqesr Fix Switch

Quantifier elimination separate roots. This is off by default. It is relevant only in OFSF for `rlqe`/`rlgqe` and for all but the outermost quantifier block in `rlqea`/`rlgqea`. For `rlqea` and `rlgqea` see Section 6.2 [Generic Quantifier Elimination], page 29. It affects the technique for substituting the two solutions of a quadratic constraint during elimination.

The following two functions `rlqeipo` and `rlqews` are experimental implementations. The idea there is to overcome the obvious disadvantages of prenex normal forms with elimination set methods. In most cases, however, the classical method `rlqe` has turned out superior.

rlqeipo *formula* [*theory*] Function

Quantifier elimination in position. Returns a quantifier-free equivalent to *formula* by iteratively making *formula* anti-prenex (see Section 5.2 [Miscellaneous Normal Forms], page 25) and eliminating one quantifier.

rlqews *formula* [*theory*] Function

Quantifier elimination with selection. *formula* has to be prenex, if the switch `rlqepnf` is off. Returns a quantifier-free equivalent to *formula* by iteratively selecting a quantifier from the innermost block, moving it inside as far as possible, and then eliminating it. `rlqews` is not available in ACFSF.

6.2 Generic Quantifier Elimination

The following variant of `rlqe` (see Section 6.1 [Quantifier Elimination], page 27) enlarges the theory by inequalities, i.e., `<>`-atomic formulas, wherever this supports the quantifier elimination process. For geometric problems, it has turned out that in most cases the additional assumptions made are actually geometric *non-degeneracy conditions*. The method has been described in detail in [DSW98]. It has also turned out useful for physical problems such as network analysis [Stu97].

rlgqe *formula* [*theory* [*exceptionlist*]] Function

Generic quantifier elimination. `rlgqe` is not available in ACFSF and DVFSF. *exceptionlist* is a list of variables empty by default. Returns a list `{th,f}` such that `th` is a superset of *theory* adding only inequalities, and `f` is a quantifier-free formula equivalent to *formula* assuming `th`. The added inequalities contain neither bound variables nor variables from *exceptionlist*. For restrictions and options, compare `rlqe` (see Section 6.1 [Quantifier Elimination], page 27).

rlgqea *formula* [*theory* [*exceptionlist*]] Function

Generic quantifier elimination with answer. `rlgqea` is not available in ACFSF and DVFSF. *exceptionlist* is a list of variables empty by default. Returns a list consisting of an extended theory and an *elimination_answer*. Compare `rlqea/rlgqe` (see Section 6.1 [Quantifier Elimination], page 27).

After applying generic quantifier elimination the user might feel that the result is still too large while the theory is still quite weak. The following function `rlgentheo` simplifies a formula by making further assumptions.

rlgentheo *theory formula* [*exceptionlist*] Function

Generate theory. `rlgentheo` is not available in DVFSF. *formula* is a quantifier-free formula; *exceptionlist* is a list of variables empty by default. `rlgentheo` extends *theory* with inequalities not containing any variables from *exceptionlist* as long as the simplification result is better wrt. this extended theory. Returns a list `{`extended *theory*, simplified *formula*`}`.

rlqegenct Switch

Quantifier elimination generate complex theory. This is on by default, which allows `rlgentheo` to assume inequalities over non-monomial terms with the generic quantifier elimination.

6.3 Linear Optimization

In the context OFSF, there is a linear optimization method implemented, which uses quantifier elimination (see Section 6.1 [Quantifier Elimination], page 27) encoding the target function by an additional constraint including a dummy variable. This optimization technique has been described in [Wei94a].

rlopt *constraints target* Function

Linear optimization. `rlopt` is available only in OFSF. *constraints* is a list of parameter-free atomic formulas built with `=`, `<=`, or `>=`; *target* is a polynomial over the rationals. *target* is minimized subject to *constraints*. The result is either `"infeasible"` or a two-element list, the first entry of which is the optimal value, and the second entry is a list of points—each one given as a *substitution_list*—where *target* takes this value. The point list does, however, not contain all such points. For unbound problems the result is `{-infinity,{}}`.

rlopt1s Switch

Optimization one solution. This is off by default. `rlopt1s` is relevant only for OFSF. If on, `rlopt` returns at most one solution point.

7 References

Most of the references listed here are available on
`http://www.fmi.uni-passau.de/~redlog/`.

[Dol99] Andreas Dolzmann. Solving Geometric Problems with Real Quantifier Elimination. Technical Report MIP-9903, FMI, Universitaet Passau, D-94030 Passau, Germany, January 1999.

[DGS98] Andreas Dolzmann, Oliver Gloor, and Thomas Sturm. Approaches to parallel quantifier elimination. In Oliver Gloor, editor, *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation (ISSAC 98)*, pages 88-95, Rostock, Germany, August 1998. ACM, ACM Press, New York.

[DS97] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209-231, August 1997.

[DS97a] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[DS97b] Andreas Dolzmann and Thomas Sturm. Guarded expressions in practice. In Wolfgang W. Kuechlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (ISSAC 97)*, pages 376–383, New York, July 1997. ACM, ACM Press.

[DS99] Andreas Dolzmann and Thomas Sturm. P-adic constraint solving. Technical Report MIP-9901, FMI, Universitaet Passau, D-94030 Passau, Germany, January 1999. To appear in the proceedings of the ISSAC 99.

[DSW98] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. A new approach for automatic theorem proving in real geometry. *Journal of Automated Reasoning*, 21(3):357-380, December 1998.

[DSW98a] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice. In B. H. Matzat, G.-M. Greuel, and G. Hiss, editors, *Algorithmic Algebra and Number Theory*, pages 221-248. Springer, Berlin, 1998.

[LW93] Ruediger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[Stu97] Thomas Sturm. Reasoning over networks by symbolic methods. Technical Report MIP-9719, FMI, Universitaet Passau, D-94030 Passau, Germany, December 1997. To appear in AAECC.

[Stu98] Thomas Sturm. Linear problems in valued fields. Technical Report MIP-9815, FMI, Universitaet Passau, D-94030 Passau, Germany, November 1998.

[SW97a] Thomas Sturm and Volker Weispfenning. Rounding and blending of solids by a real elimination method. In Achim Sydow, editor, *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling, and Applied Mathematics (IMACS 97)*, pages 727-732, Berlin, August 1997. IMACS, Wissenschaft & Technik Verlag.

[SW98] Thomas Sturm and Volker Weispfenning. Computational geometry problems in Redlog. In Dongming Wang, editor, *Automated Deduction in Geometry*, volume 1360 of Lecture Notes in Artificial Intelligence (Subseries of LNCS), pages 58-86, Springer-Verlag Berlin Heidelberg, 1998.

[SW98a] Thomas Sturm and Volker Weispfenning. An algebraic approach to offsetting and blending of solids. Technical Report MIP-9804, FMI, Universitaet Passau, D-94030 Passau, Germany, May 1998.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1):3–27, February, 1988.

[Wei92] Volker Weispfenning. Comprehensive Groebner Bases. *Journal of Symbolic Computation*, 14:1–29, July, 1992.

[Wei94a] Volker Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, FMI, Universitaet Passau, D-94030 Passau, Germany, April 1994.

[Wei94b] Volker Weispfenning. Quantifier elimination for real algebra—the cubic case. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation in Oxford*, pages 258–263, New York, July 1994. ACM Press.

[Wei95] Volker Weispfenning. Solving parametric polynomial equations and inequalities by symbolic algorithms. In J. Fleischer et al., editors, *Computer Algebra in Science and Engineering*, pages 163-179, World Scientific, Singapore, 1995.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering Communication and Computing*, 8(2):85-101, February 1997.

[Wei97a] Volker Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189-208, August 1997.

Functions

Documentation of Functions

References to Functions

Switches and Variables

Documentation of Switches and Variables

F

R

T

References to Switches and Variables

Data Structures

Documentation of Data Structures

References to Data Structures

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Short Contents

Table of Contents