REDLOG
Computer Algebra Meets Computer Logic

Andreas Dolzmann and Thomas Sturm*

Fakultät für Mathematik und Informatik
Universität Passau, D-94030 Passau, Germany
e-mail: `Andreas.Dolzmann@fmi.uni-passau.de`, `sturm@fmi.uni-passau.de`

MIP-9603

September 30, 1996

Abstract. REDLOG is a package that extends the computer algebra system REDUCE to a *computer logic system*, i.e., a system that provides algorithms for the symbolic manipulation of first-order formulas over some temporarily fixed language and theory. In contrast to theorem provers, the methods applied know about the underlying algebraic theory and make use of it. Though the focus is on simplification, parametric linear optimization, and quantifier elimination, REDLOG is designed as a general-purpose system. We describe the functionality of REDLOG as it appears to the user, and explain the design issues and implementation techniques.

* The second author was supported by the DFG (Schwerpunktprogramm: Algorithmische Zahlentheorie und Algebra)

1 Introduction

REDLOG stands for REDuce LOGic system. It provides an extension of the computer algebra system (CAS) REDUCE to a *computer logic system* (CLS) implementing symbolic algorithms on first-order formulas w.r.t. temporarily fixed first-order languages and theories. The only underlying theory currently available is that of *ordered fields*. The system is designed to be easily extensible by further theories. The REDLOG design offers the possibility to implement algorithms generically for different theories. In fact, the ordered field algorithms have already been implemented generically wherever this appeared reasonable.

The focus of the implemented algorithms is on *effective quantifier elimination* and *simplification* of quantifier-free formulas. For ordered fields there are currently the following algorithms available:

- *Negation normal form* computation. Within ordered fields we actually compute positive formulas: All negations can be encoded in the atomic formulas.
- *Prenex normal form* computation. The algorithm moves quantifiers to the outside. The number of quantifier changes is minimized, and variables are renamed appropriately where necessary.
- Several techniques for the *simplification* of quantifier-free 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 Gröbner basis computations. For the notion of simplification and a detailed description of the implemented techniques cf. [DS95b].
- CNF/DNF computation including some simplification strategies [DS95b].
- *Quantifier elimination*, i.e., computing quantifier-free equivalents for given first-order formulas. The current implementation is restricted to at most quadratic occurrences of the quantified variables. We use a technique based on elimination set ideas [Wei88], [LW93], [Wei].
- Linear *optimization* using quantifier elimination techniques [Wei94a].
- Some useful *tools* for working with formulas. For instance, selective existential/universal closure, and counting the number of atomic formulas in a formula. In addition there is support for the easy input of large systematic formulas.

We wish to point out that the range of available algorithms is strongly influenced by our major research topic in this area, i.e., quantifier elimination and simplification. Nevertheless, REDLOG is designed as a *general-purpose* computer logic system.

It is planned to extend the quantifier elimination to higher degrees. On the mathematical side, the necessary methods are known [Wei], [Wei94b].

There is an experimental implementation of a discretely valued field theory called `dvfsf`. Except for optimization and quadratic elimination there are the same algorithms available as for ordered fields.

Furthermore, some work has been done in parallelizing the ordered field optimization code under PVM on a CRAY YMP4/T3D at the Konrad-Zuse-Institut in Berlin.

2 From Computer Algebra to Computer Logic

When we originally started the implementation, our aim was to make the quantifier elimination procedures available to others for solving practical problems in physics and engineering [LS95]. The decision for taking an existing computer algebra system like REDUCE as basis has following advantages compared to a completely new implementation:

- It has shown out that CLS user interfaces require no concepts that go beyond that of CAS interfaces. Hence there is no reason for spending time in designing and implementing yet another interface. In addition, we can expect the large community of REDUCE users to be quickly familiar with REDLOG.
- The underlying system provides a reliable well-tested implementation of polynomials, which can serve as first-order terms in many languages. In addition, there is a large library of up-to-date algebraic algorithms available.
- There is no need for portability considerations. The system will simply run with all architectures and operation systems to which REDUCE is ported.

On the other hand, the underlying CAS itself can benefit from the implementation first-order formulas and corresponding algorithms. One possible application are *guarded expressions.* These occur with parametric algorithms where there are different solutions w.r.t. different conditions on the parameters. In these cases one wishes to return a list of solutions, each one *guarded* by a quantifier free condition formula in the parameters. For fixed parameters one checks for a condition that holds. The respective expression is then a solution. Guarded expressions occur, e.g., with symbolic integration, Gröbner systems [Wei92], or parametric optimization.

Another possible application is with the widespread *solve* operator for solving systems of equations and, possibly, inequalities. The solutions to such systems can be conjunctions as for $x^2 < 25$, disjunctions as for $x^2 > 25$, or more complicated formulas. The solutions are typically given as nested lists encoding DNF's. Here, the use of formulas would be more natural. At this point, one should mention that MATHEMATICA actually offers the option for printing the result as formula. Except for distributive expansion, there are, however, no algorithms operating on these formulas.

3 A User's View on the System

This section focuses on the usage of REDLOG, mainly from the algebraic mode (AM) of REDUCE. The last subsection sketches the symbolic mode interface. After loading REDLOG into REDUCE as a package, a *context* has to be selected.

3.1 Contexts

A context determines a language and a theory in the sense of first-order logic. These selections are not independent from each other. The language selection is weak in the following sense: A context does not specify which predicate symbols are allowed or prohibited. The algorithms associated with the context, however, know certain predicates. We hence speak of *known* and *unknown* predicates. Some algorithms can handle unknown predicates straightforwardly. Simplification, for instance, simply leaves unknown predicates unchanged. This behavior is quite similar to that of algebraic REDUCE operators for which no rules are known. Quantifier elimination, in contrast, would exit with an error. Schemes allowing the user to determine for unknown predicates how to behave within certain REDLOG algorithms are under consideration.

Each context is encoded into a *context identifier* as for example `ofsf`, which stands for *ordered fields* (with standard form term representation). This is actually the only context completely implemented up to now. Certain contexts may be parameterized, e.g., when computing over p-adically valued fields one might wish to fix a prime p. All following computations are performed w.r.t. the selected context until a different decision is made.

When the context is changed, formulas produced in the old context can become invalid but they need not. Certain formulas of the old context may still be meaningful in the new context. Consider for instance a formula produced in an ordered field context: If it happens to contain only variables as terms, it can be reused in an ordered set context. The same applies to formulas that can be straightforwardly rewritten in such a form as, e.g., $a - b < 0$, which would automatically be converted into $a < b$ with every access.

After fixing a context, the REDUCE functionality is extended in two ways:

1. In addition to the built-in data types such as rational functions or matrices, one can input first-order formulas.
2. There are new procedures available that apply to first-order formulas.

In the sequel, we are going to assume that the context `ofsf` is selected, which knows the binary predicates $=$, $\neq$, $\leq$, $\geq$, $<$, and $>$.

3.2 Expressions

We have extended the look-and-feel of REDUCE to first-order formulas. Invalid expressions are detected, and appropriate error messages are given.

Expression Format and Input. The format for the truth values, quantifiers, and propositional connectives is specified uniformly for all contexts. Besides the reserved identifiers `true` and `false`, there are the following operators: a unary `not`, binary infix `impl`, `repl`, `equiv`, and n-ary infix `and`, `or`. Binary prefix operators `ex` and `all` serve as quantifiers. Their first argument is a variable, and their second argument is a formula.

In general, also all atomic formulas are constructed with operators that are considered as predicates. Here again infix operators are possible. What is left to the context is determining which predicates are known and what the terms are. Furthermore, a context can impose some extra restrictions on the form of the atomic formulas. Consider for instance ofsf: The known predicates given at the end of the last subsection can all be written infix. Terms are polynomials over the integers. As an additional restriction, all right hand sides of the predicates must be zero.

The handling of the input is much more liberal than the specification of valid expressions. For the easy input of large systematic conjunctions and disjunctions the for-loops have been extended by *actions* mkand and mkor, analogous to sum or product. With the quantifiers ex and all, the first argument may be a list of variables. In ofsf the input may contain rational coefficients and non-zero right hand sides. In all these cases the input is converted to the right expression format immediately.

The ofsf context further allows the input of *chains* such as a<>b<c>d=f, which is turned into a-b<>0 and b-c<0 and c-d>0 and d-f=0. Here, only adjacent terms are considered to be related.

Simplification vs. Evaluation. Polynomials entered into REDUCE are automatically converted into some canonical form, say into distributive polynomials w.r.t. some term ordering. Canonical means that expressions which are mathematically *equal* are converted into syntactically equal forms. We refer to this conversion as *evaluation*.

The natural extension of evaluation to first-order formulas would be converting *equivalent* formulas into syntactically equal forms. Generally, this is impossible since in non-recursive structures there is no algorithm converting sentences that hold into true. Even in most recursive structures it is by no means obvious, how to obtain suitable canonical forms for open formulas. Note that such normal forms must be user-friendly and fast to compute.

Instead of evaluation, we use the weaker concept of *simplification* [DS95b]. This means, we replace formulas by equivalents that are more user-friendly though not canonical. The automatic simplification can be toggled by a REDUCE switch.

Interface Problems. Similar to other computer algebra systems, in REDUCE interpreter variables are identified with the transcendental elements occuring in rational functions. When introducing first-order formulas such an identification leads to some problems.

Firstly, for many contexts the first-order terms will be implemented as rational functions or some suitable subset. One would certainly like to identify the kernels occurring inside these terms with the interface variables. If such a kernel is quantified, any non-kernel assignment to it violates the well-formedness of the respective formula. The problem of expressions becoming bad-formed due to subsequent assignments is actually not new. This also happens with the rational function $1/x$ when assigning $x := 0$. To avoid confusion, we invalidate a formula in case of *any* assignment to a quantified variable including kernel assignments.

Another problem arises from the fact that interpreter variables interpreted as kernels are valid rational functions. They are, in contrast, not valid formulas. Hence the user is not allowed to enter things like

```
f := ex(x,g); g := x>0;
```

such that afterwards `f` be evaluated to `ex(x,x>0)`. It is, however, possible to input the above statements in reverse order.

3.3 Procedures

In order to avoid name conflicts, all REDLOG procedures and switches available in the AM are prefixed by `rl`. The procedure names are to be understood declaratively, e.g., `rlqe` stands for "apply the default procedure for quantifier elimination." Which algorithm is actually applied depends on the selected context. This pattern makes REDLOG easy to learn. Moreover, it allows to combine procedure calls to new (AM) procedures that do not depend on the context.

As usual in REDUCE we have a fairly liberal syntax including optional arguments with default values, procedures expecting either an expression of a certain type or a list of such expressions, and procedures for which the format of the return value depends on a switch.

Most of the REDLOG algorithms offer numerous options. Options are selected by setting corresponding switches.

Some procedures provide the option to protocol the progress of the computation onto the screen. We refer to this as *verbosity* output. In future versions there might be different levels of such output. It is specified, however, that there is *one* switch, namely `rlverbose`, by which all verbosity output can be turned off.

Concerning the return values, our procedures are designed to cooperate with the standard REDUCE. For example, in the ordered field context, the quantifier elimination can optionally compute sample points for existentially quantified formulas. The coordinates of such a point are returned as a list of equations because there are many built-in algorithms that operate on such lists.

3.4 Context Dependent Switches

There is a mechanism for passing the control over certain switches, say s_i, to a context c. This means when c is turned on, the current setting of the s_i is saved and then replaced by context specific values. Anyway, the user is allowed to change the setting. When the context is changed again, the current values of the s_i become the new context specific values for c, and the original values are restored. The new context can in turn take control over some switches.

This may appear to be bad style since the system modifies global settings which the user expects to be completely under his control. We need this option, however, for situations where options are not available or extremely undesirable in a certain context.

3.5 Using REDLOG from the Symbolic Mode

The package can also be used from the symbolic mode. In this case there are no optional arguments and there is no error checking. For clarity, in the names of the symbolic mode entry points the prefix `rl` is separated by an underscore from the rest of the name, e.g., `rl_qe`.

In analogy to the implementation of standard quotients, there are functions `rl_simp` and `rl_prepfof` for converting Lisp prefix to internal formula representation and back. These combined with `xread` and `mathprint` can be used for formula I/O.

4 An implementor's View on the System

REDLOG is implemented in the symbolic mode (SM) of REDUCE. The language there is the Lisp dialect RLISP, which can be compiled to machine code. The module structure of REDLOG is pictured on page 8.

4.1 The Algebraic Mode Interface

The AM interface converts the input format described in the previous section into an internal representation providing error checking. It further handles optional parameters including the substitution of default values where necessary. Finally, it calls the respective SM entry points with correct input parameters.

The internal formula representation of REDLOG is kept in the AM as a tagged form. There is a *type tag* `!*fof` (first-order formula) corresponding to the `!*sq` tag for standard quotients. This is achieved by implementing an *rtype* "logical." Using tagged forms instead of reconversion to Lisp prefix has turned out even more important than for standard quotients because, generally, there would be many first-order terms to be converted for each formula.

In addition to the type tag there is a *context tag* at every formula which is the context identifier of the context it has been created in. Comparing the context tag with the current context identifier enables the automatic conversion mentioned in Subsection 3.1. Notice that there may also be conversions that affect only the internal term representation. These are invisible for the user. Besides the conversion of the user input, which appears as Lisp prefix, and the conversion of formulas from foreign contexts, there are "equations" to be converted to "logicals" and back. Due to the special role that equations play in REDUCE, the AM interface accepts and returns equations wherever possible. The SM entry points, in contrast, insist on "logicals."

4.2 Formula Representation

The non-atomic part of the formulas is kept as Lisp prefix. So are the atomic formulas with the term representation depending on the context. The quantified variables occurring as first argument to the quantifiers are always represented

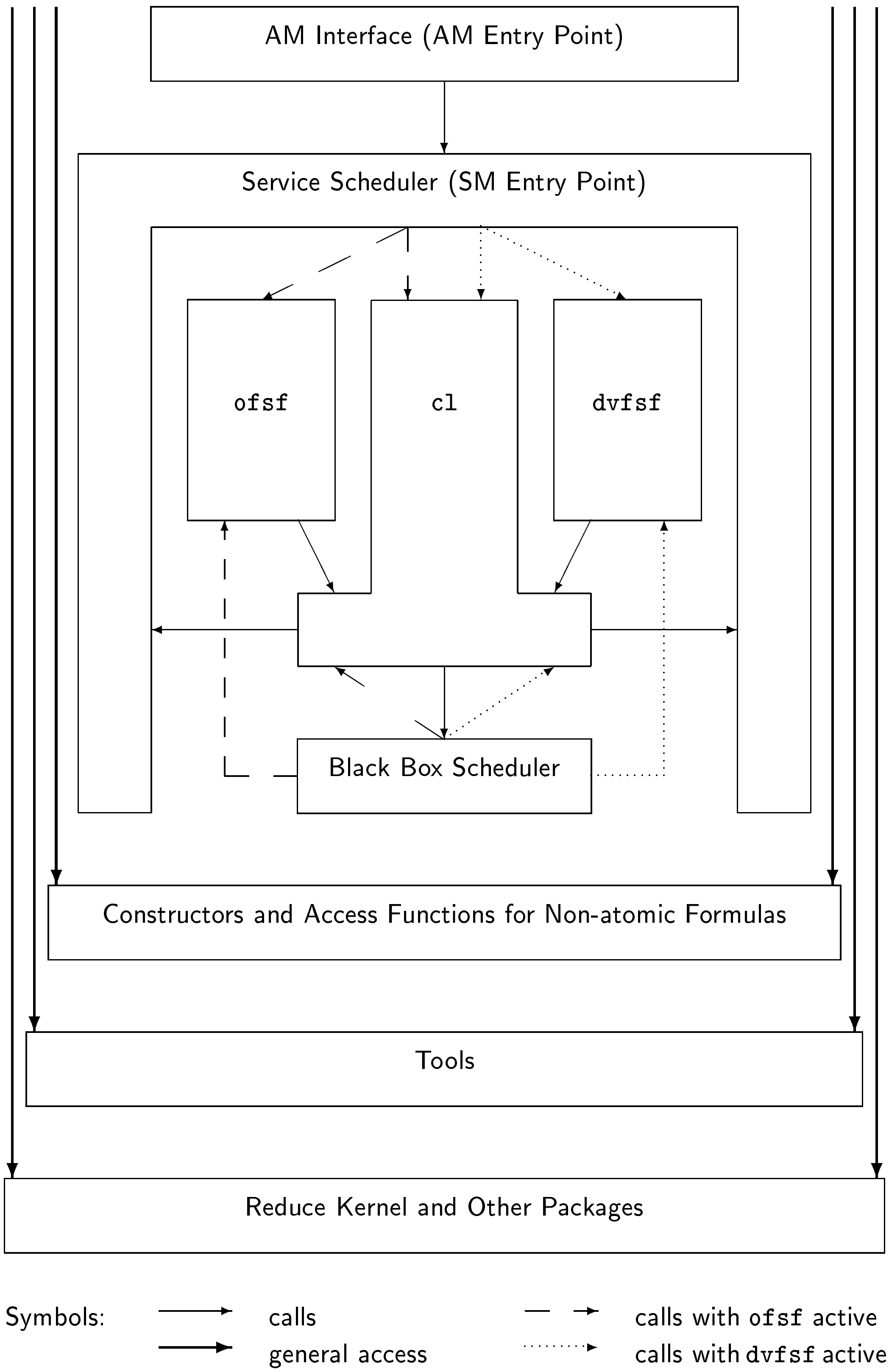

Fig. 1. The module structure of REDLOG.

as identifiers. In `ofsf` the terms are kept as standard forms. Notice that the representation of a term consisting of a single variable is different from the representation of that variable as argument to a quantifier.

The Lisp prefix specification of the non-atomic part is hidden from the whole package. There are constructors and access functions. Any direct access to formulas via Lisp operations is prohibited. We have copied this scheme to atomic formulas for the implementation of `ofsf` and `dvfsf`, and require that it has to be applied in all future context implementations.

Mind that both the type tag and the context tag used by the AM interface never enter the symbolic mode. We shall turn to this point in Subsection 4.7.

4.3 The Common Logic Module

The common logic module `cl` contains code that operates on first order formulas making no assumption about the current context.

Firstly, there are procedures that need not know the context as for instance `cl_atnum`, which given a first order formula counts the number of atomic formulas contained in it. Recall from Subsection 3.1 that recognizing the atomic formulas is no problem: Every formula that does not have a quantifier or propositional connective as top-level operator is specified as—possibly unknown—atomic formula.

Secondly, there are algorithms that split into a general part and a context dependent part. For instance, when simplifying formulas there are general rules such as deleting `false` from disjunctions or replacing disjunctions that contain `true` by the latter. On the other hand, one would design a simplifier also to replace certain atomic formulas by simpler equivalents. Such simplifications certainly depend on the context. In this situation, the general techniques are desirable for many contexts. There are several reasons for implementing them only once, and parameterizing them in some way with context dependent subalgorithms:

- New contexts can be implemented faster and safer, thus saving time for redeveloping and debugging the first order parts of the algorithms.
- All contexts will immediately profit from future algorithmic extensions and tuning of the general parts.

We say that `cl` uses *black boxes*, and accordingly we denote the code addressed by a black box as *black box implementation*. Note that there can be several generic implementations for the same task corresponding to several sets of contexts.

Local operations on atomic formulas are not the only reason for introducing black boxes. Consider, for instance, the following simplification for ordered fields: $x > 0 \land x > 1$ becomes $x > 1$. Though obviously context dependent, this applies to a complex formula.

Some black box implementations of the above kind are simply smarter variants of context independent counterparts, such as recognizing syntactically equal atomic formulas in the example. All contexts for which there is no smarter version use the context independent variant. For the reasons discussed above, the latter should be implemented only once. This leads to the existence of context independent black box implementations as a third group of `cl` procedures.

Whenever a black box implementation is required, `cl` calls the *black box scheduler*.

4.4 The Black Box Scheduler

The black box scheduler knows the current context. It performs the mapping between black boxes and black box implementations. The latter are either implemented within the active *context module* or within `cl` itself as described above. The black box scheduler is accessed only by `cl` and hence can be viewed as a part of it.

In practice, the black box scheduler is a collection of procedures which branch into the appropriate code w.r.t. the context.

4.5 The Context Modules

In the figure on page 8, there are two modules, `ofsf` and `dvfsf`, which serve as examples for context modules. We have defined a context as the specification of a language, a theory, and a term representation. There is no separation between these aspects. In particular, there is no parameterization w.r.t. the term representation: The standard form representation is quite canonical in REDUCE if one decides to keep the terms additively. Keeping the terms multiplicatively, i.e., as lists of irreducible factors leads to completely different algorithmic ideas. There is also no hierarchy of context modules, i.e., there is no ordered field context based on a field context.

The context modules are designed to cope with unknown predicates where possible. `cl` cannot even distinguish between known and unknown predicates.

For a given context, the corresponding context module contains all context dependent code. This splits into the following parts:

1. Context dependent AM interface code
2. Black box implementations
3. Complete algorithms.

The complete algorithms are context specific implementations that cannot be split into a general part and context dependent part in a reasonable economic way.

Of course, the completely implemented algorithms are not necessarily disjoint from the algorithms for which there is a `cl` implementation. Which implementations are actually used is hidden from the user who calls the `rl`-procedures from a declarative point of view (cf. Subsection 3.3). Determining which procedure to call is the task of the *service scheduler*.

4.6 The Service Scheduler

The service scheduler maps declarative computation requests to the corresponding implementations valid in the current context. Like the black box scheduler, it consists of a collection of procedures. These procedures make up the SM interface. They are also accessed by the AM interface which is implemented as a front-end to the SM interface.

The service scheduler is also called by `cl` for accessing services, which may in turn be implemented within `cl` itself. For instance, consider intermediate result simplification within a CL implementation of quantifier elimination. Though there is a simplifier implemented within `cl`, one cannot be sure that this simplifier is valid for all contexts in which the `cl` elimination code is used.

On the other hand, there are cases where `cl` should call `cl` procedures immediately. An obvious example is a recursive procedure. A more general one is a hierarchy of subprocedures built for the implementation of one or several services. In the second case, it is already possible that there is a better context dependent alternative for one of the subprocedures. It is left to the implementor to decide from case to case—we consider it impossible to give a reasonable specification for this. However, when calling `cl` code immediately, one must ensure not to address code for which there are black box implementations missing.

The context module procedures call all services immediately. There is no reason for them to call the service scheduler because they know the current context. For clarity, we have hence specified that they are not allowed to do so.

4.7 Scheduling

We have already discussed that, at any time, there is one context fixed. All input processing and all computations are performed for this context. This makes the automatic formula conversion technique described in Subsection 3.1 possible. Concerning the design of the schedulers, one should have in mind that there is always some global information on the current context available.

Black Box Scheduling. We exhibit how the black box parameterization of the `cl` functions is implemented. Sketching three possible ways to do so, each implying a certain scheduling technique, we will argue for our decision, which is for number 2:

1. A data-driven model using tagged forms.
2. A static approach using a global environment.
3. A functional model using parameter functions.

A tag is a certain identifier encoding a data type. Data type specific operations are stored in the property list of the tag. In Lisp implementations, it is a usual technique to have such a tag combined with every data object. A typical tag position is the first entry in the list representation. In our case, this would amount to keeping the context tag at the beginning of every formula. Informally spoken, the formula "knows" which black box implementations are appropriate for it. The scheduler looks at the tag, calls the implementation found there, and then returns the result.

Keeping the context information at every formula is not necessary for our task: All computations are performed w.r.t. the current context. In particular, there are never formulas combined which belong to different contexts.

In addition, this redundancy of type information even causes problems: Many of the implemented algorithms are inherently recursive. When entering a formula recursively, one would have to copy tags to subformulas for recursion, and in turn drop tags when recombining the recursion results. One could overcome this by the following technique: Save the tag globally in the beginning, and then do the recursion with tag-free formulas.

This naturally leads us to the second approach, which we actually use: All information stored in the property list of the tag in the first approach is stored in the property list of the context identifier instead. At the formulas there is no context information at all. Accordingly, the black box scheduler accesses the context identifier instead of the tag.

We consider searching the property list of the context identifier to be slow. Therefore we use a *cache* technique: The information, which is basically an *alist* mapping black boxes to implementations, is copied into a set of global variables, i.e., one variable for each black box. The functions that make up the scheduler *apply* the values of their corresponding variables to their parameters. The cache is updated whenever the context is changed.

A third technique, which we have tried in former versions, is using *parameter functions*: The names of the required black box implementations are passed to the `cl` procedures via extra parameters. The `cl` procedures address the black box implementations by *applying* the parameters. There is no black box scheduler at all. This approach has the following disadvantages:

- Due to the extra parameters the code is less readable.
- The extra parameters have to be passed down during recursion.
- When new black box algorithms and hence procedure parameters are added, all corresponding procedure calls in the code have to be adapted.

Service Scheduling. The service scheduler works the same way as the parameter scheduler does. Here, the cache technique is also adequate: Recall from Subsection 4.6 that the service scheduler does not only provide a user interface, but may also be called by `cl` procedures, possibly within loops.

In the previous paragraph we have claimed that with parameter functions there is no black box scheduler. Using parameter functions actually means leaving the black box scheduling to the service scheduler. The latter would select the appropriate service function name (service scheduling), and call it with appropriate functional parameters if it is a `cl` function (black box scheduling).

Finally, we wish to point out that with our approach the difference between services and parameters is not quite clear. There is no good reason for not viewing the parameters as services.

4.8 Tools

The tool modules contain functions required for the implementation which do not apply to logicals, but are of a more common nature. They are also used for implementing low-level code shared by different context modules, which are certainly not allowed to call each other.

5 Documentation

The documentation splits into two parts, a user manual [DS95a] and a program documentation.

The user manual is written in the GNU Texinfo format from which an online hypertext manual and a TeX document are created. There are also tools available for creating an HTML version of the document.

The program documentation is another Texinfo file, which is generated automatically from the source code: Every procedure starts with a comment block. This is an English text consisting of complete sentences. The first few sentences are specified to have specific contents. In particular, the first sentence explains the name of the procedure, e.g., in `cl_qe`, a `cl` implementation of quantifier elimination it would be: “Common logic quantifier elimination.” An alphabetic index containing the words occurring in the first sentences has shown out to be very useful with the computer algebra systems MAS and SAC. Further sentences are reserved for format descriptions of parameters, return values, and access to global variables. The comment ends with a declarative description of what the procedure does.

Acknowledgments

We wish to thank Richard Liska and Winfried Neun for many useful hints concerning the implementation. Volker Weispfenning developed a large part of the algorithmic ideas. Special thank is due to Herbert Melenk at the Konrad-Zuse-Institut Berlin who answered our questions in more than 200 mails with competence and interest.

References

[DS95a] Andreas Dolzmann and Thomas Sturm. *Redlog, a Reduce Logic Package.* FMI, Universität Passau, D-94030 Passau, Germany, preliminary edition, July 1995. User Manual.

[DS95b] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulas over ordered fields. Technical Report MIP-9517, FMI, Universität Passau, D-94030 Passau, Germany, October 1995.

[LS95] Richard Liska and Stanly Steinberg. Using quantifier elimination to test stability. Preliminary draft, 1995.

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

[Wei] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. To appear in AAECC.

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

[Wei92] Volker Weispfenning. Comprehensive Gröbner 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, Universität Passau, D-94030 Passau, Germany, April 1994. To appear in the Journal of Symbolic Computation.

[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.

This article was processed using the LaTeX macro package with LLNCS style