# Experimenting with Deduction Modulo *

Guillaume Burel

Énsiie/Cédric, 1 square de la résistance, 91025 Évry cedex, France
`guillaume.burel@ensiie.fr` `http://www.ensiie.fr/~guillaume.burel/`

**Abstract.** Deduction modulo is a generic framework to describe proofs in a theory better than using raw axioms. This is done by presenting the theory through rules rewriting terms and propositions. In CSL 2010, LNCS 6247, p.155–169, we gave theoretical justifications why it is possible to embed a proof search method based on deduction modulo, namely Ordered Polarized Resolution Modulo, into an existing prover. Here, we describe the implementation of these ideas, starting from iProver. We test it by confronting Ordered Polarized Resolution Modulo and other proof-search calculi, using benchmarks extracted from the TPTP Library. For the integration of rewriting, we also compare several implementation techniques, based for instance on discrimination trees or on compilation. These results reveal that deduction modulo is a promising approach to handle proof search in theories in a generic but efficient way.

Since proofs are rarely searched for without context, there is a strong need to be able to handle theories efficiently in theorem provers. For instance, proofs of software correction often need some flavor of arithmetic, or theories defining the data structures of the program such as chained lists. Several approaches exist to go in this direction. The first one is to design a procedure dedicated to the theory in which the proof is searched for. This would provide provers that are really adapted to the theory, but it would have the drawbacks of not exploiting the fact that theories are often built upon well-understood logics, and of being difficult to extend. In particular, combination of provers built independently for different theories would be virtually impossible. On the opposite, a second approach would be to present the theory using axioms, and to use a general-purpose theorem prover. While this method is very flexible, it is in most of the cases not efficient enough to be applied. Therefore, provers searching in theories use an in-between approach: existing general-purpose provers are combined with methods specific to the theory. SMT provers are based on this modus operandi: a prover for propositional logic (a SAT solver) is combined with a procedure specific to the theory, for instance the simplex method for linear arithmetic. SMT provers are really efficient, and are used at industrial level. Nevertheless, they suffer from the following weaknesses: they cannot prove general results, since they are restricted to ground inputs (some of them use heuristics for non-ground inputs, and there are attempts to combine first-order prover with decision

* This work was begun while the author was a post-doc fellow at the Max-Planck-Institute for Informatics in Saarbrücken, Germany.

procedures, but they are often restricted to linear arithmetic); as they handle each theory in a specific way, it is difficult to combine different theories in them, although progress has been done in that direction in the latter years, in particular thanks to the application of the Nelson-Oppen method. A solution to overcome these drawbacks is to design a framework that can be adapted to any kind of deductive system, and that handle all theories in a uniform and yet effective way. Deduction modulo [10] can be seen as such a framework. It consists in presenting a theory as a congruence over propositions, and in applying the inference rules of deductive systems modulo this congruence. The congruence is often defined by means of a rewriting system over terms and propositions. Proof-search methods derived from deduction modulo consists roughly in adding narrowing (not merely rewriting) to an existing method such as resolution or tableaux.

The study of deduction modulo has lead to strong theoretical results: any first-order theory can be presented as a rewriting system [6]; in particular, there are presentations of Peano's arithmetic [12] and Zermelo's set theory [11] with good proof-theoretical properties; it is also possible to encode higher-order systems such as Church's simple type theory or functional pure type systems as first-order theories modulo a rewriting system [9, 7]; arbitrary proof-length reductions can be achieved by working modulo a rewriting system instead of using an axiomatic presentation [5]. Nevertheless, there was no experimental results supporting the claim that deduction modulo improves indeed proof search. This was due to the fact that no implementation of proof-search methods based on deduction modulo had been developed. In [4], we have shown that integrating a resolution method based on deduction modulo into an actual prover based on ordered resolution is sound and complete, and that the given-clause algorithm, which is in most of the cases the main loop of such a prover, can be used to ease the integration. We have applied the ideas of this paper into the prover called iProver, developed by Korovin at the University of Manchester [14]. The implementation is available as a patch to iProver v.0.7 on the webpage `http://www.ensiie.fr/~guillaume.burel/empty_tools.html.en`. Here, we give the details of our implementation, and we show that using deduction modulo improves indeed proof search compared to using axioms. To do so, we choose as benchmarks problems of the TPTP library [17] that use axiom sets. Since we have to design by hand a rewriting system with good properties for each of the axiom sets, this has been done only for five of them. We also compared different ways of implementing the rewriting system. Since rewriting rules are known in advance, compiling them proved to be more efficient as soon as big terms needs to be normalized. An easy but efficient way to compile the rewriting rules is to translate them as an OCaml program that is dynamically linked to the prover.

In the next section, we present deduction modulo, and in particular the resolution calculus that has been integrated into iProver. We then detail all the technicalities of this integration in Section 2. The results of the benchmarks used to test the implementation, given in Section 3, show that deduction modulo improves the search for proofs in theories, and open perspectives given in the conclusion.

## 1 Deduction Modulo

### 1.1 Extending Deductive Systems with Rewriting

We use standard definitions for terms, predicates, propositions (with connectives $\neg, \Rightarrow, \wedge, \vee$ and quantifiers $\forall, \exists$), substitutions, term rewriting rules and term rewriting. In deduction modulo, term rewriting and narrowing is extended to propositions by congruence on the proposition structure. In addition, there are also proposition rewriting rules whose left hand side is an atomic proposition and whose right hand side can be any proposition. Such rules can also be applied to non-atomic propositions by congruence on the proposition structure. It can be useful to distinguish whether a proposition rewriting rule can be applied at a positive position or a negative one. To this end, proposition rewriting rules are tagged with a polarity and then called polarized rewriting rules. A proposition $A$ is rewritten positively into a proposition $B$ ($A \longrightarrow^{+} B$) if it is rewritten by a positive rule at a positive position or by a negative rule at a negative position. It is rewritten negatively ($A \longrightarrow^{-} B$) if it is rewritten by a positive rule at a negative position or by a negative rule at a positive position. *Term* rewriting rules are considered as both positive and negative. $\overset{*}{\longrightarrow}{}^{\pm}$ is the reflexive transitive closure of $\longrightarrow^{\pm}$. $s \overset{\mathfrak{p},\sigma}{\leadsto} t$ denotes that $s$ can be narrowed to $t$ at position $\mathfrak{p}$ with substitution $\sigma$, i.e. there exists a rewriting rule $l \rightarrow r$ such that $\sigma(s_{|\mathfrak{p}}) = \sigma l$ and $t = \sigma(s[r]_{\mathfrak{p}})$.

In deduction modulo [10], the inference rules of an existing system such as the sequent calculus are applied modulo the congruence associated with the rewriting system (term rewriting rules and proposition rewriting rules). This leads for instance to the sequent calculus modulo. In polarized deduction modulo, polarities of rewriting rules are also considered. For instance, the left and right rules for the implication in the sequent calculus become

$$\Rightarrow\vdash \; \frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, C \vdash \Delta}\; C \overset{*}{\longrightarrow}{}^{-} A \Rightarrow B \qquad\qquad \vdash\Rightarrow \; \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash C, \Delta}\; C \overset{*}{\longrightarrow}{}^{+} A \Rightarrow B$$

Proof-search methods can be derived from deduction modulo. Since variables may need to be instantiated before being rewritten, we need to perform narrowing instead of merely rewriting. In other words, we need unification instead of pattern matching. There are basically two families of proof-search methods based on deduction modulo, one extending the resolution method (ENAR [10], PRM [8]), and one extending the tableau method (TaMed [3]). In each case, the idea is to add a narrowing inference rule to the existing method.

### 1.2 Ordered Polarized Resolution Modulo

In [4], we show that it is easily possible to integrate deduction modulo into a resolution-based prover. To do so, we designed a calculus, called Ordered Polarized Resolution Modulo (OPRM$^{\succ}_{\mathcal{R}}$), and recalled in Fig. 1. Note that the ordering $\succ$ does not need to be compatible with the rewriting system $\mathcal{R}$. We proved that OPRM$^{\succ}_{\mathcal{R}}$ is complete whenever the rewriting system fulfils a criterion, namely the admissibility of the cut rule in the sequent calculus modulo.

$$\textsf{Resolution}\ \frac{P \vee C \qquad \neg Q \vee D}{\sigma(C \vee D)}\ {}^{a},{}^{b},{}^{c} \qquad\qquad \textsf{Factoring}\ \frac{L \vee K \vee C}{\sigma(L \vee C)}\ {}^{d}$$

$$\textsf{Ext. Narr.}^{-}\ \frac{P \vee C}{\sigma(D \vee C)}\ {}^{a},{}^{b},\ Q \rightarrow^{-} D \qquad \textsf{Ext. Narr.}^{+}\ \frac{\neg Q \vee D}{\sigma(C \vee D)}\ {}^{a},{}^{c},\ P \rightarrow^{+} \neg C$$

$$\textsf{Ext. Narr.}^{t}\ \frac{L \vee C}{\sigma(L' \vee C)}\ L \text{ maximal in } L \vee C,\ L \overset{\mathfrak{p},\sigma}{\rightsquigarrow} L' \text{ by a term rewriting rule, } L_{|\mathfrak{p}} \notin \mathcal{V}$$

$^{a}$ $\sigma = mgu(P, Q)$ $^{b}$ $P$ maximal in $P \vee C$ $^{c}$ $\neg Q$ maximal in $\neg Q \vee D$
$^{d}$ $L$ and $K$ maximal in $L \vee K \vee C$, $\sigma = mgu(L, K)$

**Fig. 1.** Inference rules of the OPRM$^{\succ}_{\mathcal{R}}$

We also proved that adding some simplification rules does not break this completeness. In particular, it is possible to eliminate strict subsumptions, and to normalize the clauses w.r.t. the term rewriting system. On the contrary, we gave a counter-example showing that removing tautology clauses can break the completeness. In the following, we assume that all considered rewriting systems have this cut-admissibility property. This implies in particular the confluence of the term rewriting systems.

This calculus can be easily integrated into a prover based on resolution with selection and on the given-clause algorithm by using the following remark of Dowek [8]: having a polarized rewriting rule $P \rightarrow^{-} C$, where $C$ is in clausal normal form, is the same as adding a clause $\underline{\neg P} \vee C$ where $\neg P$ is selected, apart from the fact that this clause should not be narrowed itself. Similarly, $P \rightarrow^{+} \neg C$ behave the same as $\underline{P} \vee C$. These clauses corresponding to the polarized rewriting rules are called *one-way clauses* by Dowek. To prevent such clauses to be resolved one by each other, they can be directly put into the set of active clauses in the given-clause algorithm.

## 2 Technical Details

### 2.1 iProver

iProver [14] is a first-order theorem prover developed by Korovin. It is mainly based on the Inst-Gen method: to prove that a set of clauses is satisfiable, they are made ground by instantiating all their variables with a dummy constant and passed to a SAT-solver. If the SAT-solver answers that the ground clauses are unsatisfiable, so are the original ones. If not, new instances of clauses are generated using some inference rule called Inst-Gen. In this paper, we are not really concerned with this method, although it would be interesting to study its combination with the deduction modulo framework. However, in iProver, the Inst-Gen method is combined with a resolution-based prover. We have integrated the OPRM$^{\succ}_{\mathcal{R}}$ into this part of iProver.

Since we do not use the Inst-Gen method, the choice of iProver may seem rather strange. It results from the following points:
– The most efficient provers today (Vampire [15], E [16], Spass [18], ...) are based on superposition, not only on resolution with selection. Of course, one may argue that superposition is an extension of ordered resolution with selection. Designing a calculus combining superposition and deduction modulo should not be difficult, starting from the $\mathrm{OPRM}^{\succ}_{\mathcal{R}}$. However, proving the completeness of such a calculus seems rather difficult. Indeed, as for resolution modulo, this completeness will not hold without the cut admissibility of the rewriting system. However, the standard technique to prove the completeness of superposition, namely by saturation, does not appear to be linked with cut-free proofs. The question whether one can combine the restriction of superposition with narrowing without losing completeness is therefore still open.
There is also a more technical difficulty concerning superposition-based provers. The treatment of literals by superposition is not symmetric w.r.t. their polarity: inference rules for negative literals are not the same as for positive ones, and selected literals in a clause must contain at least a negative literal if they are different from the maximal literals of the clause. Implementations of superposition exploit this asymmetry. However, we want to add one-way clauses into the prover. In these clauses, a positive literal can be selected, and it needs not to be the maximal literal w.r.t. some ordering. Just selecting this positive literal and putting the clause directly into the set of active clauses made the prover incomplete in the experimentation that we made using E, probably due to the reasons cited above.
– In the CASC-J5 competition, the first prover not based on superposition is iProver. Of course, its efficiency is largely due to the Inst-Gen method and the call to an efficient SAT-solver (namely MiniSat). Nevertheless, the data structures developed in iProver, for instance its discrimination trees, contribute to its performance, and these structures are used both by the Inst-Gen and by the resolution prover.
– iProver is written in a functional language with pattern matching, namely OCaml. Although some may argue that it can therefore not achieve the same level of performance as a prover hacked in a low-level language like C, it reveals itself to be useful in our case. Indeed, it is really easy to reflect rewriting rules into a language with pattern matching. It is therefore possible to automatically transform the input rewriting system into a program that normalizes the clauses w.r.t. it, to *compile* that program and to load it to normalize clauses. As we will see thereafter, since rewriting is compiled, this leads to real improvement in the proofs in which heavy computation is needed.

### 2.2 Input files

In practice, we do not want to write a specific parser for the polarized rewriting rules. Instead, we choose to change the semantics of the TPTP format whenever the new `--modulo` command-line argument is set to true. In that case, any formula whose role is `axiom` is understood as a rewriting rule. (It is still possible to have raw axioms by using e.g. the `hypothesis` role.) If the clause consists only of one positive literal whose main symbol is an equality, this is understood as a term rewriting rule. For instance, `cnf(plus_def_o, axiom, plus(X,o) = X).` is interpreted as the term rewriting rule $plus(X, o) \to X$. If the literal is negative, or its main symbol is not the equality, or there are more than one literal, the clause is understood as a one-way clause whose first literal is the selected one. For example,

```
cnf(all_m, axiom, ~ e(lappl(all,X)) | e(lappl(X,Y)) ).
cnf(all_p, axiom, e(lappl(all,X)) | ~ e(lappl(X,h(X))) ).
```

are interpreted as the one-way clauses $\underline{\neg\epsilon(\dot\forall @ X)} \vee \epsilon(X @ Y)$ and $\underline{\epsilon(\dot\forall @ X)} \vee \neg\epsilon(X @ h(X))$ which correspond to the polarized rewriting rules $\epsilon(\dot\forall @ X) \to^- \forall Y.\ \epsilon(X @ Y)$ and $\epsilon(\dot\forall @ X) \to^+ \neg\neg\epsilon(X @ h(X))$ (see [8]). The special case of the reflexivity axiom $X = X$ is also treated as a one-way clause and not as a rewriting rule.

## 2.3 Clause Generation by Narrowing

To perform the Ext. Narr.$^-$ (resp. Ext. Narr.$^+$) inference rule, we add a rewriting rule $P \to^- C$ (resp. $P \to^+ \neg C$) as a one-way clause $\underline{\neg P} \vee C$ (resp. $\underline{P} \vee C$). To this end, we need to select $\neg P$ (resp. $P$) in the clause, and put the clause directly into the set of active clauses, before the main loop of the given-clause algorithm is performed. Then, applying Resolution with one of these one-way clauses simulates Ext. Narr.$^\pm$. In iProver, selected literals in a clause are just a list of literals that is attached to the clause. Selecting a literal in a clause consists therefore simply in calling `assign_res_sel_lits` with the singleton list containing the left-hand side of the rule. Inserting the clause in the active set is done as it would be for a normal clause: adding the clause into the unification index (using the selected literal) and tagging the clause as active.

Implementing the Ext. Narr.$^t$ in iProver is more difficult. Indeed, iProver does not have a special inference rule such as paramodulation to handle equality. If equalities are present, iProver only add the axioms that define equality in the current signature. Therefore, we need to add a paramodulation inference rule ourselves. Fortunately, some data structures to do so were already present for the resolution inference rule. For instance, active clauses are indexed using a non-perfect discrimination tree [13]. To add narrowing by a term rewriting rule, we add a new index, `rewrite_index_ref`. Only term rewriting rules are added into this index. Given a term $t$, the index will provide all candidate rewriting rules, *i.e.*, only rules whose left-hand side can possibly be unified with $t$. Then, for all candidates $l \to r$, one tries to unify $l$ with $t$, and if it is the case, one returns $\sigma(r)$ where $\sigma$ is the substitution computed during the unification. However, this is not sufficient, since term narrowing should perform at any depth in the term $t$. Therefore, we implemented a data structure for contexts, allowing one to go inside terms, and if a term cannot be narrowed at one position $\mathtt{p}$, narrowing is tried on all position directly below $\mathtt{p}$. Note that by doing so, all clauses that could be generated by Ext. Narr.$^t$ are not, since we do not go below a position if narrowing was successful. However, we generally assume that the term rewriting system is sufficiently well-formed (in particular, as stated above, it is assumed to be confluent) so that it does not break the completeness of the prover.

## 2.4 Simplifications

As recalled before, some simplifications that are compatible with standard ordered resolution break the completeness of $\mathrm{OPRM}^{\succ}_{\mathcal{R}}$. For instance, tautologies cannot be eliminated. Because they break the completeness, or we do not know if they preserve it, we have to switch off the following options of iProver: `--instantiation_flag`, `--schedule`, `--prep_prop_sim`, `--ground_splitting`, `--res_to_prop_solver`, `--res_orphan_elimination`; `--res_lit_sel` is set to `kbo_max`. There is no flag in iProver to turn off tautology elimination, so we changed the source code to prevent their elimination whenever the new `--modulo` flag is set to `true`.

In $\mathrm{OPRM}^{\succ}_{\mathcal{R}}$, clauses can be narrowed using the term rewriting system, hence generating new clauses, but we have shown that they can also be normalized, *i.e*, replaced by their normal form. Indeed, adding the demodulation simplification rule ($C$ is simplified to $D$ if $C \longrightarrow D$ by the term rewriting system) does not break the completeness, and repeatedly applying this simplification eventually leads to a normal form of the term, assuming it exists.

There are several way to perform the normalization of the clauses. We compared the following ones, that can be selected using the `--normalization_type` parameter:

**none** No simplification is performed, clauses have to be rewritten using Ext. Narr.$^t$, generating new clauses.

**interp** Rewriting rules are translated into OCaml closures performing the pattern matching: by structural induction on the left-hand side of the rule, a function is built that matches its arguments w.r.t. the left-hand side and returns a substitution:

```
let rec term_to_subst = function
| Term.Fun(f, f_args, _) -> (function
   Term.Fun(g, g_args, _) when f = g ->
     List.fold_left2
      (fun sub t1 t2 -> merge_subst (term_to_subst t1 t2) sub)
      (Subst.create ()) f_args g_args
  | _ -> raise No_match)
| Term.Var(var,_) ->
    let sub = Subst.create () in
      fun t -> Subst.add var t sub
```

If this function is successful, the obtained substitution is applied to the right-hand side. If not, one tries another rewriting rule. If no rewriting rules can be applied at that position, one tries the same method below in the term.

**dtree** Thanks to the implementation of $\mathsf{Ext.Narr.}^t$, there is already a data structure that helps in retrieving rewriting rules whose left-hand side can be unified with some term. Since pattern-matching is stronger than unification (if a term matches a pattern, then the term and the pattern can be unified), the same structure can be used to get candidates for matching. Here also, one needs to test rewriting deeply in the term.

**pipe** Rewriting rules are known statically once the input file is parsed, since $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$ does not generate new rewriting rules. Therefore, they can be compiled to improve their efficiency. A simple way to compile them is to translate them into a OCaml program using pattern matching. For instance, the rules $f(X, g(X)) \to h(X)$ and $f(h(X), Y) \to Y$ are translated into the code

```
let match_term = function
  Fun("f",[x0; Fun("g",[x1])]) when x0 = x1 -> Fun("h",[x0])
| Fun("f",[Fun("h",[x0]); y0]) -> y0
| _ -> raise No_match
```

This translation is fully automated. Then, there is no need to implement an efficient pattern-matching algorithm, since it is the one of OCaml that will be used. This `match_term` function is added into an OCaml source file `pipe_iprover.ml`. There, it is called by a tree-traversal that tries to apply it at each position of the term. Note that it is easy for the user to change the rewriting strategy, since one only has to change the traversal in `pipe_iprover.ml` before launching iProver. The file contains a main loop that does the following: it waits a term on the standard input, normalizes it and put the result on the standard output. This file is then compiled, and the resulting program is run. iProver then communicates with it through UNIX pipes. Terms are expected to be passed using the marshalling function of OCaml. This implies that the version of OCaml used for compiling iProver must be the same as the one for compiling `pipe_iprover.ml`.

**plugin** As for pipe, an OCaml program is compiled, but it is loaded using the dynamic loading library Dynlink of OCaml, which is available for native compilation since version 3.11 for most platforms: the `match_term` function is added into a file `plugin_iprover.ml` which is compiled as a dynamic library and loaded. The main function of the compiled plug-in changes only a reference to a normalization function, pointing it to the function that does the normalization using `match_term`. iProver has just to use the new reference to get the normalization function. Here again, the normalization strategy can be easily modified by the user by changing `plugin_iprover.ml`.

**size_based** Compilation costs time. It is therefore not clear that the two previous options are more efficient, in particular when only small terms are rewritten. This last normalization method decides to launch the compilation (plugin style) only when a term whose size reaches some threshold needs to be normalized. For smaller term, the dtree method is used. The threshold can be changed using the `--normalization_size` command-line parameter.

## 3 Benchmarks

### 3.1 Comparison with Other Calculi

We first test whether $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$ really improves proof search compared to standard ordered resolution with selection using "normal" axioms. As we need to switch off some simplifications in order $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$ to be complete, we compare it to the following calculi:

**Ordered resolution, same restrictions as OPRM:** in this case, the same options are given to iProver as when $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$ is tried, the only difference is that the `--modulo` flag is switched off, the axioms being therefore considered as normal clauses instead of rewriting rules.

**Ordered resolution, default options of iProver:** in this case, the default options of iProver are used; only the Inst-Gen prover is turned off.

**Full iProver** iProver is launched with its default options; in particular, the Inst-Gen prover is combined with the resolution prover.

We may also have compared it to another prover, in particular a prover based on superposition such as SPASS or E. Notwithstanding, this seems unfair, since the resolution prover of iProver is written in OCaml whereas other provers are written in C, and contain a lot of low-level optimizations, leading to more efficient executables.

To perform a benchmark, we need a set of problems to test. We therefore need some theories, and some problems related to these theories. The TPTP library [17] provides a number of axiom sets, each of them used in several problems. We could have tried to consider each of these axiom sets as a theory. The main difficulty is that for each of them, we have to define an equivalent rewriting system for which cut admissibility holds, in order to guarantee the completeness of $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$. There exists a procedure that transforms a set of axioms into a rewriting system with this property [6]. However, first, this procedure may not terminate, and second, it never was implemented, although we did write some prototype which showed us that the procedure produces systems that are too big to be usable. We therefore had to design rewriting systems and prove their cut admissibility by hand. Consequently, we only tried five theories, named after their TPTP v4.0.0 axiom-set files. We tested all the problems of the TPTP library that use these axiom sets. The rewriting systems we designed to present these theories are given at `http://www.ensiie.fr/~guillaume.burel/empty_tools.html.en`. We considered **ANA001**, axioms defining the analysis (limits) for continuous functions, **BOO001**, axioms defining a ternary boolean algebra (boolean algebra with a ternary multiplication function), **FLD001**, axioms defining ordered fields, **SET001** and **SET002**, axioms defining a weak set theory using resp. predicates or function symbols to define unions, intersections, differences and complements. We ran each problem with a time-out of 60 s: first using the rewriting system in $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$, second using the axiom set of the TPTP in resolution with the same restriction as $\mathrm{OPRM}_{\mathcal{R}}^{\succ}$, third in resolution with default options and fourth using iProver in its whole. All tests were performed under Linux 2.6 on a four-core Intel® Core™ i3 CPU M330 at 2.13GHz.

**Table 1.** Comparison of Different Calculi on Problems Extracted from the TPTP Library. #: number of solved problems; %: percentage in the problem set corresponding to the theory; $\bar{t}$: average time to find a proof for the solved problems.

| | ANA001 | | BOO001 | | FLD001 | | SET001 | | SET002 | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # (%) | $\bar{t}$ | # (%) | $\bar{t}$ | # (%) | $\bar{t}$ | # (%) | $\bar{t}$ | # (%) | $\bar{t}$ | # (%) | $\bar{t}$ |
| OPRM | 3 (75) | 11.41 | 3 (100) | 0.01 | 40 (29) | 0.95 | 15 (100) | 0.01 | 8 (100) | 0.01 | 69 (42) | 1.05 |
| restricted resolution | 0 ( 0) | NA | 0 ( 0) | NA | 23 (17) | 2.85 | 15 (100) | 4.05 | 5 ( 63) | 8.06 | 43 (26) | 3.88 |
| default resolution | 1 (25) | 25.34 | 1 ( 33) | 25.46 | 40 (29) | 13.55 | 15 (100) | 0.96 | 7 ( 88) | 22.99 | 64 (39) | 12.00 |
| full iProver | 1 (25) | 0.18 | 1 ( 33) | 0.42 | 42 (31) | 4.69 | 15 (100) | 0.17 | 7 ( 88) | 7.11 | 66 (40) | 3.79 |

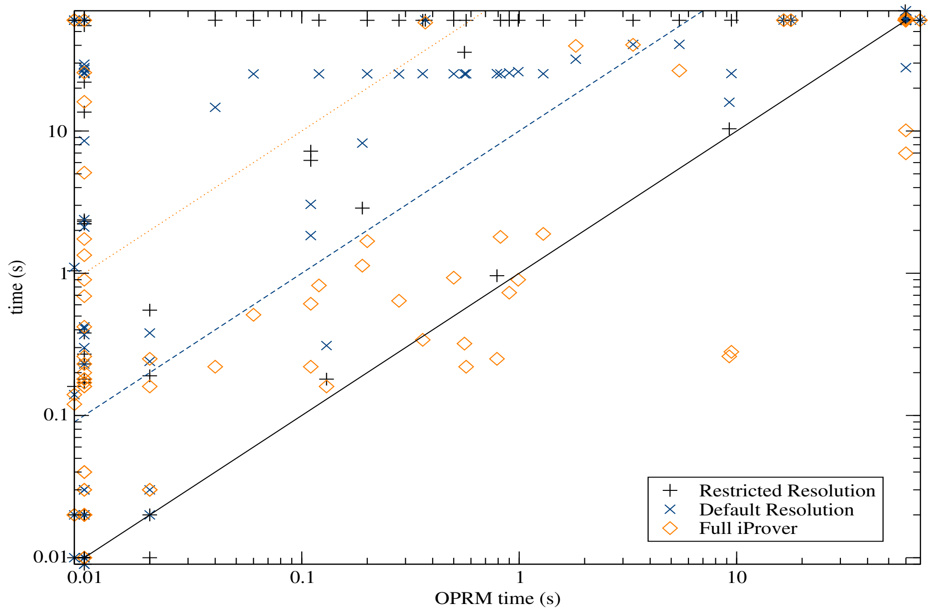


**Fig. 2.** Comparison of Different Calculi on Problems Extracted from the TPTP Library. The x-axis gives the time taken by OPRM$^{\succ}_{\mathcal{R}}$, the y-axis by the other calculus.

The results are summarized in Table 1 and represented graphically in Figure 2. The time taken for a given problem by OPRM$^{\succ}_{\mathcal{R}}$ is compared to the time taken by the other calculi. Since the scale is logarithmic, for all points above the dashed line, OPRM$^{\succ}_{\mathcal{R}}$ is 10 times faster than the other calculus, and for all points above the dotted line, 100 times faster. As we can see, OPRM$^{\succ}_{\mathcal{R}}$ is always at least as efficient as restricted or default resolution, and in most of the cases at least 10 times better. This was expected, because having proved the cut admissibility for the considered rewriting system implies that the theory is consistent, and the prover does not try to find a contradiction in the theory. A more surprising result is that using iProver in its whole is only rarely much better than using OPRM$^{\succ}_{\mathcal{R}}$. This means that the gain of using OPRM$^{\succ}_{\mathcal{R}}$ relative to using ordered resolution is comparable to the gain obtained by combining it with the Inst-Gen method (including the use of an efficient SAT-solver).

## 3.2 Comparison of Rewriting Implementations

In this section, we want to compare the different techniques that can be used to perform the normalization of the clauses w.r.t. the term rewriting system. To have a better control of the required amount of normalization, we do not rely on "real" problems of the TPTP, but on three families of problems crafted by hand. We first use tests requiring only normalization. The first one consists of proving that $n+n = 2\times n$ in Peano's arithmetic, *i.e.*, given $\underline{n} \overset{\text{def}}{=} s^n(o)$, we have to prove $\underline{n}+\underline{n} = s(s(o))\times\underline{n}$ modulo the rewriting system

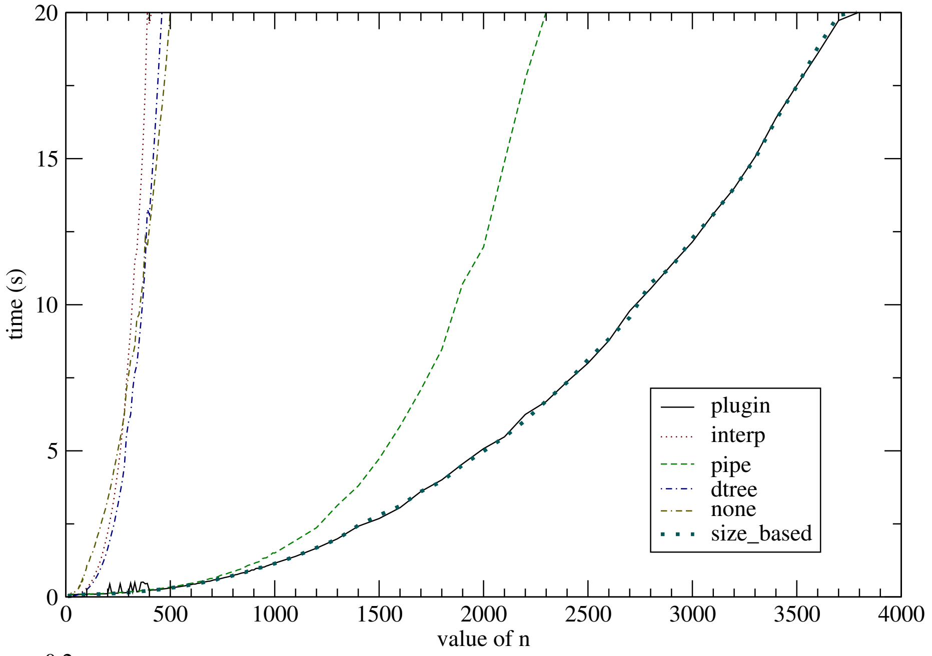
$$\begin{array}{lll} s(X)+Y \to s(X+Y) & o+Y \to Y & X = X \to^+ \neg\bot \\ s(X)\times Y \to (X\times Y)+Y & o\times Y \to o & \end{array}$$

The second one consists in proving the same theorems, but using Church's integers in a $\lambda$-calculus with explicit substitutions. This calculus, similar to $\lambda_\upsilon$ [1], is defined using binary operators for application $(\cdot @ \cdot)$ and substitution $(\cdot[\cdot])$, unary operators for lambda abstraction $(\lambda)$, unit substitution $(/)$ and substitution lifting $(\Uparrow)$, De Bruijn indexes represented by 1 and *sc*, and the shifting substitution $\uparrow$. Then, given $\overline{n} \overset{\text{def}}{=} \lambda(\lambda(sc(1) @ (\cdots (sc(1) @ 1))))$, we have to prove $(+ @ \overline{n}) @ \overline{n} = (\times @ \overline{2}) @ \overline{n}$ modulo the rewriting system

$$\begin{array}{ccc} \lambda(A) @ B \to A[/B] & (A @ B)[S] \to A[S] @ B[S] & 1[/A] \to A \\ (\lambda(A))[S] \to \lambda(A[\Uparrow S]) & sc(N)[/A] \to N & 1[\Uparrow S] \to 1 \\ sc(N)[\Uparrow S] \to N[S][\uparrow] & sc(N)[\uparrow] \to sc(sc(N)) & 1[\uparrow] \to sc(1) \\ X = X \to^+ \neg\bot & \times \to \lambda(\lambda(sc(1) @ 1)) & \end{array}$$
$$+ \to \lambda(\lambda(\lambda(\lambda((sc(sc(sc(1))) @ (sc(sc(1)) @ sc(1)) @ 1)))))$$

Arguably, these tests do not reflect reals proofs, since they consists only of normalization, and no inference is performed. To have a test mixing both normalization and inference, we used an encoding of instances of the Syracuse conjecture, *i.e.*, given an $n$, we tried to prove that by dividing $n$ by 2 if $n$ is even, and multiplying it by 3 and adding 1 if it is odd, and reiterating the process, 1 is reached eventually. This was encoded by proving $syracuse(\underline{n})$ modulo:

$$\begin{array}{rr} syracuse(X) \to^+ \neg\neg syracuse'(X, parity(X)) & parity(s(o)) \to false \\ syracuse(s(o)) \to^+ \neg\bot & parity(o) \to true \\ syracuse'(X, true) \to^+ \neg\neg syracuse(\frac{1}{2}(X)) & \frac{1}{2}(s(s(X))) \to s(\frac{1}{2}(X)) \\ syracuse'(X, false) \to^+ \neg\neg syracuse(\times 3+1(X)) & \frac{1}{2}(s(o)) \to o \\ parity(s(s(X))) \to parity(X) & \frac{1}{2}(o) \to o \\ \times 3+1(s(X)) \to s(s(s(\times 3+1(X)))) & \times 3+1(o) \to s(o) \end{array}$$



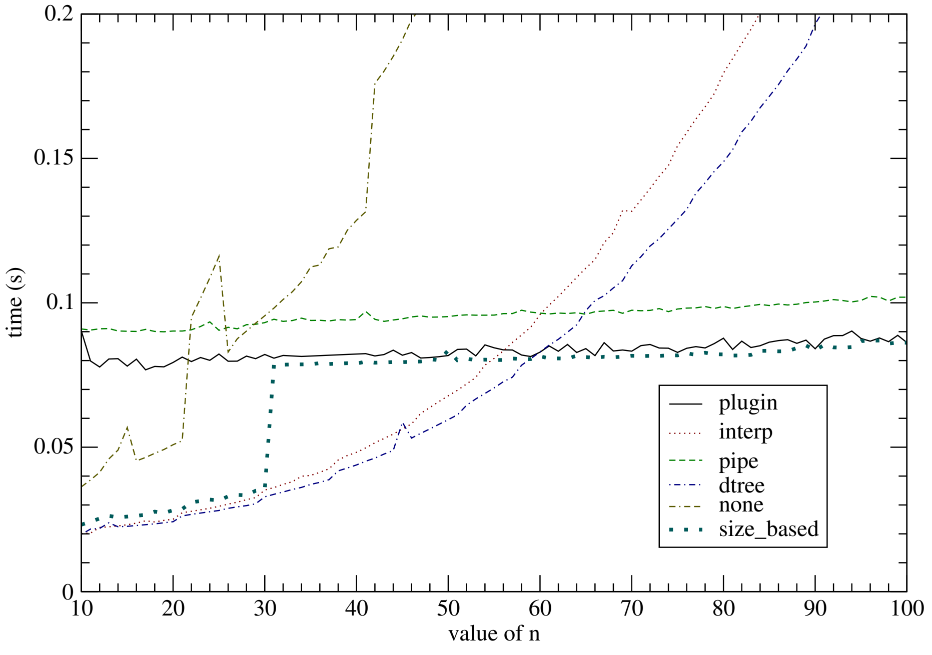


**Fig. 3.** Comparison of different techniques for implementing normalization: Proving that $n + n = 2 \times n$ in Peano's arithmetic. Values of $n$ from 10 to 4000, and zoom from 10 to 100.

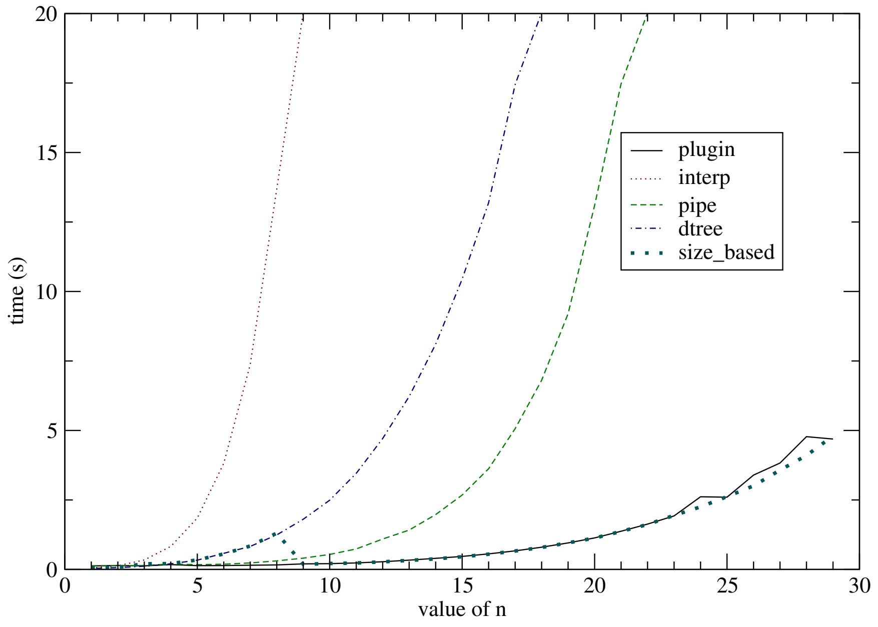


**Fig. 4.** Comparison of different techniques for implementing normalization: Proving that $n + n = 2 \times n$ with Church's integers.

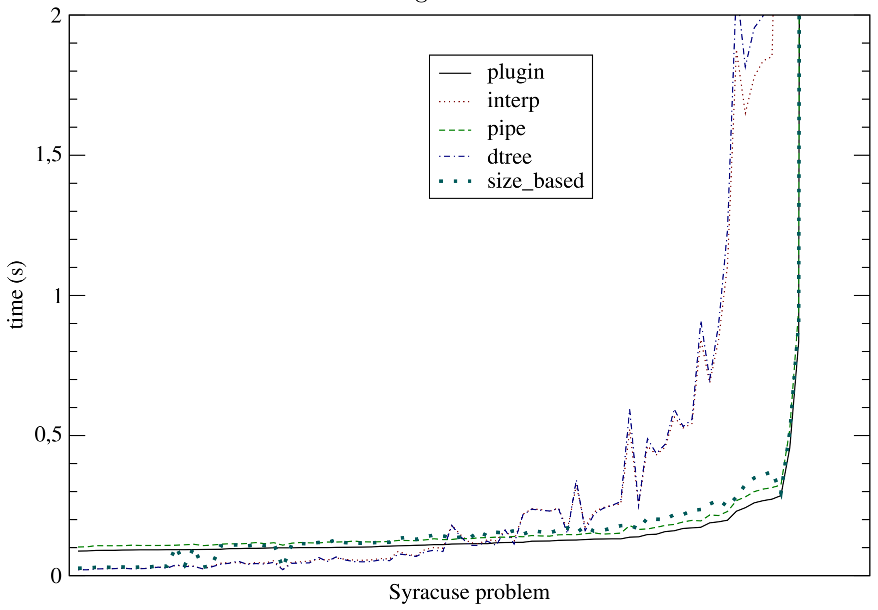


**Fig. 5.** Comparison of different techniques for implementing normalization: Instances of the Syracuse conjecture.

The results are represented in Figures 3 to 5. As expected, compilation of the rewriting rules leads to much better results when heavy computation is needed. We also note that using Unix pipes degrades performance for large terms, compared to using a plug-in. The invocation of the OCaml compiler costs time. There is a threshold of approximately 0.07 s in Peano's arithmetic and for the Syracuse problem. Using the size_based method seems a fair choice, since it behaves, as expected, like dtree on small inputs and like plugin on large ones. However, the term-size threshold for launching compilation depends on the problem: it should be set greater for Peano's arithmetic, and smaller for Church's integers. This is due to the fact that normalizing a term of a given size requires more applications of the rewriting system for Church's integers than in Peano's arithmetic, so that the gain obtained by compiling the rules is greater for the formers. The rewriting systems of these tests are not meant to be good at reasoning about arithmetic, their purpose is to compare the different normalization techniques; the difference that were enlightened for these tests should occur for any rewriting system.

## Conclusion

The benchmarks presented in the previous section demonstrate that using a rewriting system instead of axioms improves proof search, and that compiling the rewriting system is efficient as soon as big terms are rewritten. One could argue that one should not have used raw axioms, but a saturated set of clauses instead. There are two remarks to be made: A saturated set of clauses, if viewed as one-way clauses, can be seen as a rewriting system with cut admissibility. Conversely, the one-way clauses corresponding to a rewriting system with the cut admissibility does *not* need to be saturated w.r.t. the inference rules of the system to guarantee the completeness; they are therefore less numerous, and the completeness does not depend on the clause ordering.

A point that strongly needs to be studied is the automatic transformation of an axiomatic presentation into an equivalent rewriting system with cut admissibility. As mentioned above, a procedure exists, but its implementation showed that it would be impractical. A way to improve it would be to work on the remark above, namely that saturated set of clauses can be seen as cut-admitting rewriting systems. Note that once the set is saturated w.r.t. some ordering, it can be used with another ordering without breaking the completeness.

Another issue is to study whether extending superposition with narrowing preserves completeness. If not, we should search extra criteria that would imply it. This is crucial since we plan to integrate deduction modulo into today's most efficient first-order provers such as Vampire, E, or SPASS.

Last, compiling rewriting rules to improve first-order provers is not a new idea, but it was put aside because the compilation time is too long when the compiler needs to be called for each new rewriting rule generated by the system. Here, such a problem is not present, since rewriting rules are known in advance, *i.e.* once the input has been read. Moreover, our approach for the compilation of the rewriting rules is reminiscent of the normalization by evaluation technique [2]. To really have NbE, we should translate not only the rewriting rules into OCaml programs, but also the terms to be normalized themselves. It is not clear whether this would really improve proof search, but it should be tested.

## References

1. Benaissa, Z., Briaud, D., Lescanne, P., Rouyer-Degli, J.: $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. Journal of Functional Programming 6(5), 699–722 (1996)
2. Berger, U., Eberl, M., Schwichtenberg, H.: Normalization by evaluation. In: Prospects for Hardware Foundations. LNCS, vol. 1546, pp. 117–137. Springer (1998)
3. Bonichon, R., Hermant, O.: A semantic completeness proof for TaMed. In: Hermann, M., Voronkov, A. (eds.) LPAR. LNCS, vol. 4246, pp. 167–181. Springer (2006)
4. Burel, G.: Embedding deduction modulo into a prover. In: Dawar, A., Veith, H. (eds.) CSL. LNCS, vol. 6247, pp. 155–169. Springer (2010)
5. Burel, G.: Efficiently simulating higher-order arithmetic by a first-order theory modulo. Logical Methods in Computer Science 7(1:3), 1–31 (2011)
6. Burel, G., Kirchner, C.: Regaining cut admissibility in deduction modulo using abstract completion. Information and Computation 208(2), 140–164 (2010)
7. Cousineau, D., Dowek, G.: Embedding pure type systems in the lambda-pi-calculus modulo. In: Ronchi Della Rocca, S. (ed.) TLCA. LNCS, vol. 4583, pp. 102–117. Springer (2007)
8. Dowek, G.: Polarized resolution modulo. In: Calude, C.S., Sassone, V. (eds.) IFIP TCS. IFIP AICT, vol. 323, pp. 182–196. Springer (2010)
9. Dowek, G., Hardin, T., Kirchner, C.: HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. Mathematical Structures in Computer Science 11(1), 1–25 (2001)
10. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. Journal of Automated Reasoning 31(1), 33–72 (2003)
11. Dowek, G., Miquel, A.: Cut elimination for Zermelo's set theory (2006), available on authors' web page
12. Dowek, G., Werner, B.: Arithmetic as a theory modulo. In: Giesl, J. (ed.) RTA. LNCS, vol. 3467, pp. 423–437. Springer (2005)
13. Graf, P.: Term Indexing, LNAI, vol. 1053. Springer (1996)
14. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P. (eds.) IJCAR. LNAI, vol. 5195, pp. 292–298. Springer (2008)
15. Razianov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Communications 15(2-3), 91–110 (2002)
16. Schultz, S.: System description: E 0.81. In: Basin, D.A., Rusinowitch, M. (eds.) IJCAR. LNCS, vol. 3097, pp. 223–228. Springer (2004)
17. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
18. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE. LNCS, vol. 5663, pp. 140–145. Springer (2009)