Case Splitting in an Automatic Theorem Prover for Real-Valued Special Functions

James P. Bridge and Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England

{jpb65,lp15}@cam.ac.uk

22 February 2012

Abstract

Case splitting, with and without backtracking, is compared with straightforward ordered resolution. Both forms of splitting have been implemented for MetiTarski, an automatic theorem prover for real-valued special functions such as exp, ln, sin, cos and $\tan^{-1}$. The experimental findings confirm the superiority of true backtracking over the simulation of backtracking through the introduction of new predicate symbols, and the superiority of both over straightforward resolution.

Contents

1 Introduction

Backtracking, or depth-first search, is one of the most elementary techniques used in Artificial Intelligence. It is the basis of the Prolog programming language and many automated reasoning technologies, including satisfiability checkers (SAT and SMT) and theorem provers based on analytic tableaux. The obvious exception is resolution theorem proving, which works by saturating a set of disjunctive clauses derived from axioms and a negated conjecture, terminating with success if it detects a contradiction. But even some resolution theorem provers (starting with SPASS [15]) support backtracking.

Logically speaking, if (in the course of a proof search) we have established the disjunction $C \vee D$, then the cases C and D can be considered separately. If a contradiction can be derived in each case, then a contradiction follows from $C \vee D$. (For resolution, C and D must have no variables in common; however, a free-variable tableau calculus can cope with shared variables.) If further disjunctions deduced in the course of the search are treated similarly, then we are generating a search tree. The resolution method can deal with disjunctions directly: instead of deducing a contradiction from C, it can deduce D from $C \vee D$ (by essentially the same proof) and then deduce a contradiction from D. The drawback of relying on unmodified resolution is that in a derivation with many steps of this kind, the intermediate disjunctions become long. Resolution heuristics favour short disjunctions, making this type of derivation difficult to find. Case splitting could allow resolution to solve much harder problems than before, provided appropriate criteria can be found for splitting a disjunction. To justify the inevitable overheads, both C and D should somehow be non-trivial.

Backtracking is complicated to implement in the context of a resolution theorem prover [4]. Many of the data structures (normally designed for a saturation role) need to be saved and restored. Moreover, backtracking must be "intelligent": if the C case succeeds without using C, then the contradiction is independent of $C \vee D$ and there is no need to consider the D case. The search should now ignore all irrelevant splits and backtrack to the nearest relevant one in the search tree. The potential performance gains are obvious. Non-chronological backtracking is now regarded as essential to the DPLL method used in satisfiability checkers.

A simple alternative to backtracking is to introduce a new predicate symbol (taking no arguments) for each split [9, 11]. For example, the disjunction $C \vee D$ can be replaced by the two disjunctions $C \vee p$ and $D \vee \neg p$, where p is a new predicate symbol. A simulation of splitting, retaining many of its advantages, can then be obtained through resolution alone: no major modifications to control or data structures are required. Given the clause $C \vee p$, if resolution can refute C, then it will deduce p, and at the next step it can deduce D, continuing our case-analysis. This type of splitting automatically yields intelligent backtracking: given $C \vee p$, if C turns out to be irrelevant then there is no need to make use of $D \vee \neg p$. This lightweight splitting is generally regarded as inferior to backtracking, perhaps under the principle of no pain no gain, but experimental evidence is limited. The use of predicate symbols here is related to their use to abbreviate subformulas in order to obtain compact clause forms [10].

We have implemented both forms of case splitting for MetiTarski, an automatic theorem prover for real-valued special functions [1]. MetiTarski is a first-order resolution theorem prover augmented with computer algebra algorithms and decision procedures, and equipped with axiom files specifying a variety of upper and lower bounds for several transcendental functions. We outline our implementations of both forms of splitting and present data to compare their efficacy. We hope that our methods and results will be of use to others despite our prover's unusual architecture and application domain.

Paper outline. We introduce (Sect. 2) MetiTarski, describing its overall architecture, and describe splitting. We then describe implementations of splitting both without (Sect. 3) and with (Sect. 4) backtracking. Experimental results appear in Sect. 5, followed by conclusions (Sect. 6).

2 Background

We work in the context of resolution theorem proving [2]. But the theorem prover we are developing, MetiTarski, differs from typical resolution theorem provers in many respects. A brief overview of MetiTarski appears below, along with a more detailed introduction to case splitting, both with and without backtracking.

To begin, we need some standard definitions. A resolution theorem prover operates on a large formula in conjunctive normal form, which is invariably conceived as a set of clauses as follows:

An *atomic formula* $P, Q, \ldots$ has the form $t = u$ or $t \leq u$, where t and u are terms. MetiTarski interprets all terms as ranging over the real numbers.

A *literal* is an atomic formula or its negation. MetiTarski regards $t < u$ as abbreviating the literal $\neg(u \leq t)$ and accepts the abbreviations $t \neq u$, $t \geq u$ and $t > u$.

A *clause* is a finite set of literals, interpreted as their disjunction. We typically write clauses as logical formulas such as $\neg P_1 \vee \neg P_2 \vee Q_1 \vee Q_2$ instead of sets such as $\{\neg P_1, \neg P_2, Q_1, Q_2\}$. The empty clause denotes the formula $\perp$, contradiction. A set of clauses is interpreted as their conjunction.

The clause C is called a *component* of a clause $C \vee D$ if C is nonempty and if C and D have no variables in common.

A *ground* term, literal or clause is one containing no variables.

More detailed and complete definitions can be found elsewhere [2, 11].

2.1 MetiTarski

MetiTarski [1] is a resolution theorem prover for inequalities involving real-valued special functions. It shares much code with Joe Hurd's Metis prover [5]. It uses computer algebra code (both internal to itself and external packages, QEPCAD in particular) to simplify algebraic formulas and to decide polynomial inequalities. It also relies upon axioms describing the behaviour of special functions, such as the following facts about logarithms:

$$\frac{(1+5x)(x-1)}{2x(2+x)} \leq \ln x \leq \frac{(x+5)(x-1)}{2(2x+1)}$$

Because QEPCAD does not recognise the division operator, MetiTarski also requires axioms relating division to multiplication, such as this one:

$$\neg(X \leq Y \cdot Z) \vee X/Z \leq Y \vee Z \leq 0$$

The standard resolution procedure, equipped with such axioms and assisted by suitable weights and other heuristics, replaces special function inequalities by polynomial inequalities that are deleted if they prove to be inconsistent, thereby taking the search closer to the empty clause.

The following examples demonstrate MetiTarski's capabilities and limitations. The following formula is proved in about 30 seconds:

$$x \neq 0 \land |x| < \pi/2 \Longrightarrow \sin(x)^{-2} - \frac{1}{7} < (x+3\pi)^{-2} + (x+2\pi)^{-2} + (x+\pi)^{-2} + x^{-2} + (x-\pi)^{-2} + (x-2\pi)^{-2} + (x-3\pi)^{-2}$$

It is the instance for $n = 3$ of a formula from Mitrinovic's *Analytic Inequalities* [8, p. 245] (recall that $\csc x = 1/\sin x$):

$$\csc^2 x - \frac{1}{2n+1} < \sum_{k=-n}^{n} (x - k\pi)^{-2}$$

Proved in 9.5 seconds is a formula concerning the correctness of a Chua circuit [3]:

$$0 \le x \le 289 \Longrightarrow 2.84 - .063e^{-.019x} - 1.77e^{.00024x}\cos(.0189x) + .689e^{.00024x}\sin(.0189x) < 5$$

The substantial runtimes are misleading, because the actual MetiTarski proof search takes under two seconds for each problem. The remaining time is spent in QEPCAD, and this is typical of proofs that require lengthy runtimes. However, improving the performance of the computer algebra aspect of MetiTarski is not the focus of this paper.

2.1.1 Algebraic Literal Deletion

QEPCAD and related computer algebra software are the key to MetiTarski's crucial heuristic: *algebraic literal deletion* [1, §4.2]. An algebraic literal may be deleted from a clause provided it is inconsistent with other algebraic assertions in its context, which consists of the negations of the clause's other literals combined with the active clause set. For example, suppose we have a clause asserting $x^2 > 2$. Given the clause $x + x^4 \le 3 \lor x < 0 \lor C$, MetiTarski will delete the literal $x + x^4 \le 3$ because

$$\exists x\, (x \ge 0 \land x^2 > 2 \land x + x^4 \le 3)$$

simplifies to false. Logically speaking, MetiTarski introduces the clause $x + x^4 > 3 \lor x < 0$ to express the contradiction found by computer algebra, finally resolving it with the given clause to delete the literal and derive $x < 0 \lor C$.

Case splitting complicates this procedure because some of the available clauses may be specific to certain splits. It is necessary to keep track of which clauses are available at any given point. During backtracking, the set of available clauses will change.

2.1.2 Product Splitting

MetiTarski employs heuristics to isolate occurrences of special functions, in particular when they are multiplied together [1, §4.1.3]. Consider a literal of the form $tu \le t'$, where t contains special functions and t' does not. It seems natural to divide both sides by u in order to isolate the term t.

Given a clause of the form $tu \le t' \lor C$ (and analogously for $t' \le tu \lor C$), a special inference rule creates three new clauses (one for each possible sign of u):

$$t \le t'/u \lor u \le 0 \lor C$$
$$t'/u \le t \lor u \ge 0 \lor C$$
$$0 \le t' \lor u > 0 \lor u < 0 \lor C$$

The product tu is thereby split into its separate parts, t and u, which can be further isolated as the proof proceeds. If both of these terms contain special functions, literals such as $t \leq t'/u$ and $u \leq 0$ can be seen as separate MetiTarski problems to be tackled independently. Some time ago, we identified examples of such clauses that could not be refuted in a reasonable amount of time but where the two halves could be refuted easily if they were manually separated. It was on the basis of such examples that we decided to investigate case splitting.

2.2 Splitting With Backtracking

The first implementation of case splitting in a resolution theorem prover based on the superposition calculus is SPASS [14], dating from the mid-1990s. The current version of SPASS implements a *labelled splitting* calculus due to Fietzke and Weidenbach [4], which we have followed. Given a clause $C \vee D$ and a component C, a split starts a derivation from C, and if that leads to the empty clause, continues with D (assuming C was actually used).

It should be noted that their work has different objectives from ours. Fietzke and Weidenbach emphasise the aim of obtaining effective decision procedures for certain fragments of first-order logic; to ensure termination, they prefer that C has fewer variables than $C \vee D$. Along with most of the automatic theorem proving community, they attach a high importance to completeness. Our problem domain of special-function inequalities has no complete proof methods, and we are aware of no interesting decidable fragments. Our sole objective is to increase the number of proofs found in a given amount of time. (We have arbitrarily chosen 60 seconds.)

The labelled splitting calculus is based upon straightforward principles. Each split is identified by a unique integer, and each clause is labelled with a set of such integers to identify every split that has contributed to its derivation. In a typical inference, the label of the conclusion is the union of the labels of the premises. The depth-first search is managed by a data structure called the *split stack*. When backtracking occurs, careful bookkeeping is necessary to ensure that all data structures contain assertions relevant to the current case; obsolete information must be deleted, but also, subsumed clauses (previously identified as being redundant) may need to be reinstated. Full details are available in the paper [4], including proofs that the calculus is sound and complete.

There remains the crucial question of when to split a clause. Fietzke and Weidenbach are vague on this point, saying that a clause should be split provided “the first split part has a ‘sufficient’ reduction potential for other clauses” [4, p. 33]. We insist that both split parts refer to special functions, and are ground formulas. (The latter condition trivially ensures that there are no variables in common, and most of our special-function formulas are ground anyway.) Thus, we split a clause if both parts are likely to require substantial derivations in their own right and not merely for the sake of checking routine side-conditions.

2.3 Without Backtracking

In order to avoid the pervasive architectural changes outlined above, a number of authors have advocated splitting without backtracking [9, 11]. New propositional symbols replace the clause labels, and straightforward modifications to the standard resolution heuristics eliminate the need for stacks and explicit backtracking. We decided to follow Riazanov and Voronkov [11], who describe a number of refinements in great detail. Nivelle [9] describes a variation on the method, with a focus on producing a saturated set of clauses.

Lightweight splitting might be expected to be inferior to backtracking, and therefore our implementation was a conceived as a minimal-effort pilot project. If it improved MetiTarski's success rate, then we would proceed to an implementation of backtracking. Otherwise, we would drop the idea altogether.

3 An Implementation of Splitting Without Backtracking

The implementation of lightweight case splitting involved several milestones.

1. The basic inference rule, introducing new predicate symbols as labels and modifying algebraic literal deletion (as described above) to correctly identify the clauses available in the current context. A clause is available provided each of its labels is also a label of the clause being simplified.

2. Heuristics for the order in which cases are attempted. In particular, unless care is taken, both cases will be worked on simultaneously.

3. The naming heuristic [11], an optimisation for the important situation where a particular literal takes part in more than one split.

It was easy to implement, requiring modest and local changes. Coding started in December 2010. Case splitting started working shortly afterwards, though initially with poor results. Within a month, good heuristics had been implemented, delivering a significant improvement to the success rate.

3.1 Basic Strategy

In general, case splitting identifies a component of a clause, dividing the clause into two nonempty parts having no variables in common. We are only concerned with ground clauses (no variables at all) and require each case to involve a special function inequality. In other words, the predicate symbol must be $\leq$ (recall that $<$, $\geq$, $>$ each abbreviate instances of $\leq$) and one of the operands must involve a special function (any function other than $+$, $-$, $\times$, $/$, abs).

Lightweight splitting requires the use of predicate symbols to identify the cases. These need to be distinguished from other literals, and in particular, it would be senseless to split a clause at such a label. Riazanov and Voronkov [11] require a component to contain no such literals. As mentioned in Sect. 2.1.1 above, when performing algebraic literal deletion, it is necessary to identify which clauses are available in the context of the given clause, C. Availability is easy to check: a clause is available provided its case labels are a subset of those of C. As special cases, we recognise a clause that depends on no splits (perhaps part of the original conjecture), and a clause involved in exactly the same combination of cases.

Riazanov and Voronkov [11] distinguish between binary splitting and hyper splitting. The former splits a single component from a clause, leaving the remainder of the clause to be further split if possible; the latter immediately splits a clause among multiple components. It is not obvious why hyper splitting would perform better than binary splitting, but in our experiments, its advantage was clear.

Case splitting in MetiTarski can be illustrated by a diagram somewhat different from Riazanov and Voronkov's [11]:

$$\begin{aligned}&L_1 \vee \ldots \vee L_n \vee C \mapsto \\ &L_1 \vee p_1,\ \ldots,\ L_n \vee p_n,\ C \vee \neg p_1 \vee \ldots \vee \neg p_n\end{aligned}$$

The literals $L_1, \ldots, L_n$ each involve the relation $\leq$ (positive or negative: in other words, $\leq$ or $<$) and a special function. Clause C must contain exactly one other such literal,[1] and the entire given clause must be ground. The literals $p_1, \ldots, p_n$ are the case labels and must be freshly generated, but see "splitting with naming" in Sect. 3.3.

3.2 Literal Selection: Parallel versus Serial Derivations

In its basic form, the resolution inference rule combines two clauses, $C \vee A$ and $D \vee \neg B$, to yield the conclusion $(C \vee D)\sigma$, where σ is the most general unifier of the literals A and B. *Ordered* resolution restricts the application of this rule to eliminate redundancy from derivations. Generally, the literals must be maximal in their clauses according to the Knuth-Bendix ordering; a *selection function* can impose further criteria, which can depend upon the literal's sign. We can also modify the ordering itself, at the level of literals. Bachmair and Ganzinger [2] describe some of the many variations on these ideas.

Without splitting, a derivation from $C \vee D$ can work on any of the literals in this clause. Even if the restrictions of ordered resolution allow only one literal at a time to take part in resolution steps, we can envisage the derivation as involving clauses of the form $C_i \vee D_i$, for $i = 1, \ldots,$ with C_i derived from C and D_i derived from D. Derivations for C and D can be constructed in parallel. We derive the empty clause from $C \vee D$ if C_k and D_k are both empty for some k.

Recall that a basic splitting step replaces the clause $C \vee D$ by the two clauses $C \vee p$ and $D \vee \neg p$, where p is a new predicate symbol. If both clauses are made active in the normal way, we could expect to see a similar parallel derivation, with clauses of the form $C_i \vee p$ and $D_i \vee \neg p$ being generated. Now, if C_k and D_k are both empty, then we have derived p and $\neg p$ and immediately obtain the empty clause.

To develop derivations in parallel looks undesirable. In general, many generated clauses do not contribute to any final proof, so let us assume for the sake of argument that a derivation based on C will be fruitless in that no C_i will be empty. Let us further assume that this fruitless search will actually die out, yielding only heavy clauses that will be ignored. A proof can perhaps still be found using other clauses. However, any work devoted to $D \vee \neg p$ will be wasted: even if we manage to derive $\neg p$, it is only useful if we can derive p from $C \vee p$, contrary to our assumption that derivations from C are fruitless. For similar reasons, in the absence of splitting, no derivation from $C \vee D$ can lead to a proof and any inference steps using literals of D will be wasted.

The alternative to parallel search is a serial search, simulating backtracking. This involves arranging things such that, given the clauses $C \vee p$ and $D \vee \neg p$, the latter clause is temporarily blocked. We can arrange this by ensuring that, given $D \vee \neg p$, the only selected literal is $\neg p$. Then nothing can happen with this clause until we have derived p, that is, after a successful derivation from $C \vee p$. With this approach, no work is done on D until we have obtained success with C.

We can realise either parallel or serial derivations through appropriate literal selection. Under the standard Knuth-Bendix ordering, case labels (which are predicate symbols) are inherently considered to be smaller than all other literals, so the default ordering will only select a case label when no other literals are available. This yields parallel execution. To obtain serial execution, we can modify the ordering so that negated case labels ($\neg p$) are greater than all other literals. A clause containing a negative case label will not be able to take part in any resolution step except to cancel out one of those labels. For MetiTarski, serial derivations proved to be overwhelmingly

[1]The way a clause's literals are ordered as $L_1, \ldots, L_n$ is described in Sect. 3.4 below.

superior to parallel ones.

3.3 Splitting With Naming

It is natural to think of the case label p in $L \vee p$ as an abbreviation for L. With naming, case labels are not freshly generated. Instead, we use an injective naming function N from literals to labels.

$$\begin{aligned}&L_1 \vee \ldots \vee L_n \vee C \mapsto \\ &L_1 \vee N(L_1),\ \ldots,\ L_n \vee N(L_n),\ C \vee \neg N(L_1) \vee \ldots \vee \neg N(L_n)\end{aligned}$$

(Our definitions are simpler than those given in Riazanov and Voronkov [11, §3.1] because we perform splitting on the basis of ground literals rather than general components.)

Crucially, there is no need to generate a clause of the form $L_i \vee N(L_i)$ more than once. If such a clause has been generated and has produced a successful derivation, then we now have the clause $N(L_i)$. In effect, we have learned the fact $\neg L_i$ and do not need to consider L_i again. Experiments prove that this one heuristic is particularly powerful.

The question then arises, why do we require C to contain a non-trivial literal (one involving $\leq$ and a special function)? In this decision, we deviate from Riazanov and Voronkov [11] and abandon the possibility to apply naming to this final literal. The answer, typically, is empirical: if we arrange that C contains only trivial literals, then results are much worse. A possible explanation is that this policy requires one more split for every clause that contains trivial literals. For instance, the clause $\ln x > 2 \vee x \leq 0$ could be split, with C as $x \leq 0$. It is unsurprising that these additional splits could complicate the search space.

3.4 Ordering the Cases

The choice of serial rather than parallel processing of cases makes their order significant. One component of the split will be developed while the others are set aside. The Knuth-Bendix order is the obvious basis for ordering the components. During ordinary resolution, the ordering allows only maximal literals to take part. It may seem natural to order the components of a split using Knuth-Bendix inverted, so that maximal components are preferred. However, experiments show that it is best to prefer minimal components. With hindsight, the reason for this is clear: splitting removes the largest literals (which under Knuth-Bendix implies the most complicated literals) from consideration. Unless the first (and therefore simplest) case is successful, the remaining cases will never be taken up and will be permanently removed from the search.

Splitting with naming also benefits from this choice. Simple literals will be considered first, and they are more likely to occur repeatedly than complicated literals. Repeated occurrences of a literal will allow the corresponding splits to be skipped altogether.

3.5 Miscellaneous Tweaks

In early experiments, splitting did not appear to be promising. It did not significantly increase the number of theorems proved; although some proofs ran faster, others ran slower or failed. Case splitting apparently needed to be limited, and we experimented with the following parameters:

- Limiting the total number of splits. We tried limits of 10, 50 and into the hundreds.

- Attaching a weight to case labels. Like other resolution theorem provers, MetiTarski draws clauses from a priority queue giving preference to "simpler" clauses. The clause $t < 0$ is obviously simpler than $t < 0 \vee p$, because the literal p essentially abbreviates an arbitrarily difficult additional problem hidden in the clause $\neg p \vee C$.

Experiments showed that varying these parameter settings affected the success rate significantly, with different settings yielding different sets of solutions. But as the full architecture took shape, further experiments demonstrated that the optimal case label weight was zero. Moreover, the success rate increased monotonically with the limit on the number of splits. In other words, both parameters became unnecessary. The keys to success have been described above: serial derivations, splitting with naming and ordering the cases.

At the end of this experiment, splitting (without backtracking) was yielding a significant improvement in the success rate, compared with no splitting. Some problems could now be proved in seconds that had not been proved before, and for others, the runtime decreased by orders of magnitude. On this basis, we proceeded to a full implementation of splitting.

4 An Implementation of Splitting With Backtracking

Our implementation carefully follows the guidelines presented by Fietzke and Weidenbach [4], and we use the same terminology.

4.1 Split Stack and Labelled Clauses

Each split is a simple split between two cases. The first case, or left split, consists of a single literal containing a special function. The second case, or right split, consists of the rest of the clause being split. The right split clause may be further split when it is reintroduced to the clause set on refutation of the left split. On splitting, the right hand clause is removed from the clause set and stored, as it will be reintroduced if the left hand clause is involved in a chain of inferences leading to the empty clause. The right hand clause is stored on a stack, the *split stack*, together with further information needed for backtracking.

A pair of integers is assigned to each split. An odd integer identifies a left split, while the next greater even integer identifies the corresponding right split. These integers are attached to all clauses that depend on that split: not only the left and right split clauses themselves, but also any clauses derived from them. A set of such integers is called a *label*. A clause's label identifies all the splits on which it depends.

These integers act as shorthand for the literals removed from the clause by the split. Given that the two halves of the split have no variables in common, any inference step involving labelled clauses would be equally correct (ignoring ordering constraints) if the clauses were replaced by expanded clauses with the labels replaced by the corresponding hidden literals. This allows the conversion of proofs that involve splitting to equivalent proofs that do not involve splitting, as described in Sect. 4.5.

The integer associated with the active split is added to the *split stack*, which encodes the search tree. This integer—initially odd, later replaced by its even counterpart—defines the *stack level*.

In addition to the right split clause, the negation of the left split clause is added to the stack, as it may be added to the clause set if the left branch gets refuted. These form the set of *blocked clauses* described by Fietzke and Weidenbach [4]. In our implementation, the left split is added to the stack without being negated and is only negated after the right branch of the split is entered; thus it can be stored as a single clause, while negating a clause yields a separate clause for each literal. With the present implementation the left split is a single literal anyway, but the distinction could become important if in future multi-literal splits are made. The left split is refuted when the empty clause is derived. This empty clause will have a label showing the splits it is dependent on and these dependencies must be passed on to the blocked clauses. This empty clause is also stored on the stack and is referred to as the *leaf marker*.

4.2 Reinstating Subsumed and Unsimplified Clauses

In Fietzke and Weidenbach's implementation [4], the stack also stores clauses that have been deleted as redundant because of clauses that themselves depend on that split level. We take a different approach to deleting and restoring clauses. In our implementation, deleted clauses are saved in a list separate from the stack.

Inference steps often result in a clause being replaced by a simplified form, or (in the case of subsumption) even being deleted. Where the premises of such steps contain labelled clauses indicating a dependence on splits, backtracking may require the step to be undone and the original clause restored. For example, if the left split clause subsumes another clause, after the empty clause is found (refuting the left case), the subsumed clause must be returned to the clause set.

In SPASS [4], deleted clauses are stored in the split stack at the level of the clause that led to the deletion (determined by the highest split label in its label set). When, upon backtracking, any clause of the same level is removed, all deleted clauses at that level are reinstated unless they depend on splits that have been removed. The process is complicated by the need to check for clauses in deleted sets as well as in the current set when finding from which levels clauses should be reinstated.

The SPASS approach has the disadvantage of reinstating clauses that may still be redundant, as Fietzke and Weidenbach note in their paper [4]. Also, it requires searching the entire stack for clauses dependent on newly deleted stack levels, which is more complicated than searching a separate list. Saving the deleted clauses at different stack levels also means that they must be processed as each stack level is deleted, which is less efficient than processing them all at the end of a multi-level backtrack.

Our implementation keeps a separate list of "conditionally deleted" clause pairs, consisting of subsuming and subsumed clauses. (In fact, we only store the label of the subsuming clause.) When a clause is simplified using literal deletion (Sect. 2.1.1), the unsimplified clause is stored as the subsumed clause with the simplified clause stored as the subsuming clause. Clause pairs are not kept if the subsuming clause's label is a subset of the label of the subsumed clause, as any split deletion that removes the subsuming clause will also remove the subsumed clause. A special case is where the subsuming clause is a top level clause (one that is not dependent on any splits); such a clause will have an empty label, which is a subset of any other label, including another empty label.

Following backtracking, where one or more split levels are deleted, the list of deleted clauses is processed. Any subsumed clause whose label is not a subset of the set of levels on the current stack is permanently deleted. Of the surviving subsumed clauses, any whose subsuming clause's label is not a subset of the set of levels on the current stack is reinstated, and the pair is removed from the deleted clause list.

At every split, the parent clause is subsumed by each of its children. We could simply allow the subsumption mechanism to detect this. By treating this case specially, saving the parent clause on the split stack, we eliminate these unnecessary and expensive subsumption checks.

This procedure avoids the need for multiple reinstatements and deletions during backtracking. The clause labels are kept as ordered lists, so checking whether one label is a subset of another takes linear time. The overhead of maintaining a list of deleted clauses is acceptable. It is offset by the reduction in the number of clauses added to the clause set, since redundant clauses are never reinstated.

4.3 Backtracking

Clause labels are lists of integers kept in decreasing order. The first element, the greatest integer, is the most recent split. The remaining integers identify earlier splits on which the clause relies. If the labelled clause is empty, then the series of assumptions embodied in the splits that are encoded in the clause label is inconsistent. Now backtracking occurs: the most recent open split is returned to and the alternative case tried. If there is no split to return to (that is, with its right split available), then the original clause set has been shown to be inconsistent: the conjecture is a theorem.

The odd/even labelling scheme provides an easy way to detect cases that are yet to be tried. Odd integers are left branches, so there is still a right branch to be explored. Even integers are right branches, so there are no further cases to consider at that level. Fietzke and Weidenbach [4] describe the process in detail.

If the label of an empty clause starts with a series of even numbers, then these are skipped over until an odd number is reached. Call it l. This is the level to which backtracking must take place. This is the process of *right collapse*. (If there are no odd numbers in the label, then right collapse continues all the way to the top level, indicating that a proof has been found.) During backtracking, splits with levels greater than l are removed from the split stack. The original parent clause is restored; clauses that arose from the split are deleted; clauses that had been subsumed and are no longer redundant are restored.

In addition, any odd numbered splits above l which are not part of the label of the empty clause may be deleted from the top of the stack down to the first even numbered stack level. Undoing needless splits is referred to as the *branch-condense* process. After backtracking, the process of considering the right split case is referred to as *enter-right*.

4.3.1 Deleting Clauses From the Active and Waiting Sets

At the end of the backtracking process, a number of levels will have been deleted from the stack. As a consequence, some clauses will need to be deleted, and some previously deleted clauses will need to be reinstated.

There are two sets from which clauses might be deleted.

- The *active* clauses. This set is saturated (closed under all possible inferences).
- The *waiting* clauses. These have yet to take part in the proof.

MetiTarski has inherited data structures from the theorem prover Metis. Neither the active nor the waiting sets are implemented as simple sets or lists. The active set consists of several data structures, which are designed for efficient subsumption or rewriting or the exploitation of unit clauses and so on. For each of these structures, each clause contained in them must be checked and deleted if its label is no longer a subset of the split labels still on the split stack. The waiting set is implemented as a priority queue using a purely functional data structure. Arbitrary deletions from this queue are impossible. We therefore adopt lazy deletion, maintaining a list of deleted levels against which clauses are checked, and possibly discarded, as they are removed from the queue.

4.3.2 Complications of Backtracking

A clause label is a set of integers, as outlined in Sect. 4.1, identifying the splits on which the clause depends. In an inference step, the conclusion relies on all the clauses that form the premises. The label of the conclusion is the union of the labels of the premises. In our implementation, each label is an ordered list; unions are formed by merging and deleting duplicates.

The more integers in a label, the more likely it is that the clause will be deleted on backtracking, along with any clauses derived from it. This means that the use of clauses that are not strictly needed may be detrimental. In MetiTarski, this is most likely to occur when calling an external decision procedure, passing a list of algebraic clauses that form the current environment. The label of the simplified clause that results must be the union of the labels of all the clauses passed to the decision procedure, but not all such clauses may be needed. The problem is that the decision procedure is slow and determining a minimum set of clauses is too costly. In an experiment, we tried calling decision procedure first with just the top-level clauses and only if that failed trying the full set of algebraic clauses. In practice, even this simple scheme increased the overall proof search time because of the additional decision procedure calls.

The implementation of backtracking was complicated by some data structures that MetiTarski had inherited from the original Metis theorem prover. These structures were designed to provide efficient rewriting and exploitation of unit clauses but were not designed to identify the equations or unit clauses involved in any particular simplification. As rewriting is not a major element of MetiTarski proofs, the most straightforward solution was to exclude any clauses with non-empty labels from the rewrite structure. This approach is sound but may miss some potential rewrites.

4.4 Negated Left Splits and Right Splits

On refutation of the left split, the right branch of the split is entered. Two new clauses are released: the negation of the left split and the right split clause. In general, the negation of the left split consists of n clauses if the left split contains n literals. Additionally, if the left split contains variables, these will become new Skolem constants; then the negated left split clause is best discarded. In MetiTarski, the left split clause is a single ground literal L, so its negation is simply $\neg L$.

These two clauses could be added to the waiting set in the same manner as other new clauses or restored clauses. We considered restricting the scope of the negated left clause, mindful of the danger that it could generate a large, fruitless derivation. Experimentation demonstrated that simply adding it to waiting gave the best results. The right split clause is treated differently: it is given priority over clauses in the rest of the waiting set, which are queued according to weight. The idea is to accord priority to finish splits that have already been half proved. This yields better results than simply adding the clause to the normal waiting queue, though the difference in performance is not large.

4.5 Generating Proofs Without Splits

MetiTarski delivers machine-readable proofs in TSTP format [13]. In theory, these could be checked by a separate tool. We remove case splitting steps from the proofs to make them easier to verify. The clause labels encode sufficient information for this to be done.

One approach is to create a map between the odd numbered split labels and the corresponding right case split. These need to be kept until the end of the proof, and it is simpler to avoid split label reuse when the stack grows again after backtracking. All labelled clauses can then have the corresponding right split clauses reinstated. In the case of a single split, for example, the proof steps leading from the left split clause C to the empty clause become proof steps leading from the parent clause $C \vee D$ to the right split clause, D. Unfortunately, the number and size of clauses to be added can be large.

We use a novel approach, taking advantage of the left split clause being a ground literal. Instead of starting from a parent clause $L \vee D$ where D could be large (or could have a long label implying several additional clauses), the first step of the subproof is the assumption of the top level tautology $L \vee \neg L$. The first clause of the existing subproof L is then considered as the left split of $L \vee \neg L$ and not as the left split of $L \vee D$; only one literal, $\neg L$, needs be added to the clauses of the refutation of L, and the final clause is $\neg L$. Finally, the right hand split is obtained by resolution between $\neg L$ and $L \vee D$.

A complication is that the label of the left split clause is different in the proof search from that needed in the proof reconstruction. That is, for the purposes of proof reconstruction, the left split clause is made a simple split from a new top level tautological clause consisting only of the left split and its negation, with a zero length label. The label of the left split itself is then only a single integer. In theory, this same clause with a single integer could also be used in the proof search in place of the left split arising from the original parent clause, which has a much longer label. In practice, it was found that better results were obtained with the longer labelled clause. The long label in the proof search is beneficial, as it restricts the search space and ensures that the left split clause is tied to the correct parent clause and is deleted if any of the associated splits are undone.

In the implementation of proof reconstruction, both labels are combined. As the shorter label is contained within the longer label, we can identify it by a simple trick: negating the integers that correspond to the shorter label.

4.6 Additional Refinements

Results were disappointing at first, and we explored a variety of refinements to the basic approach in the hope of improving them.

4.6.1 Correcting Distance Measures

The resolution loop in MetiTarski uses the classic *given clause* algorithm [6], where the given clause is selected from a priority queue. The priority is determined by a clause weighting system based on such things as the number and kind of symbols within the clause. To ensure that all clauses are eventually considered, clause weights also contain a term measuring the derivation distance of the clause from the original axioms and negated conjecture. With ordered resolution, the distance measure is not necessary for completeness, but it is still beneficial, discouraging newly-generated clauses from swamping the search space.

Prior to the introduction of splitting with backtracking, the distance of each new clause was calculated by simply incrementing a value after each given clause had been processed. With splitting with backtracking, two issues arose with the simple approach.

- What distance should be assigned to clauses that are deleted and then reinstated?
- Since the right split is always dealt with after the left split has been refuted, clauses in the right branch have unfairly large distance values and hence weights. (The distance counter increases throughout the refutation of the left branch.) Therefore, these clauses are considered later than they should be.

The solution to these problems was to modify the code so that clauses carry a distance as well as a label. When clauses are reinstated, they retain their original distance. When a case split occurs, both left and right splits are given the same distance. When the right branch is entered, the distance counter is reset to that of the right split, and used to assign distances to new clauses.

4.6.2 SOS WEIGHTING

The *set of support* (SOS) contains those clauses whose proofs involve at least one clause arising from the negation of the conjecture. The SOS strategy requires at least one of the premises of each inference to be drawn from the set of support. This forbids resolution steps within the set of axioms, which is (we hope!) consistent. "The set of support strategy avoids seeking a proof, a demonstration of unsatisfiability, within that subset which is assumed to be itself satisfiable." [16, p. 537]

Though the SOS strategy is complete for standard resolution [2], it is not complete in combination with ordered resolution, as used in MetiTarski. Nevertheless, SOS can improve results, especially when there are many axioms [7, Sect. 4.6].

With MetiTarski, experiments showed that a strict SOS strategy allowed hardly any proofs to be found. A less extreme approach has been implemented. Instead of excluding all new clauses outside the SOS, such new clauses are added to the waiting set, but with their weight multiplied by a user defined factor (currently 2.0). Applying such an SOS weighting factor is generally beneficial in the case of splitting with backtracking. (Experimental results are presented below.) In other situations (no splitting, splitting without backtracking), it seems to be detrimental.

5 Experimental Results

5.1 Methodological Issues and Soundness

We compare various approaches according to the number of theorems proved. It is legitimate to ask, how do we know that a proof is valid, in view of the absence of competing systems to use for validating or refuting results? Ideally, MetiTarski proofs should be fed into an automatic and independent proof checker, but no such tool currently exists. We forego any claim to absolute correctness, but have adopted measures to detect some incorrect proofs.

For testing and evaluation, we use a set of approximately 800 problems. Of these, more than 100 have never been proved. None of the problems are actually false, as far as we know, although many are impossible to prove using our methods. Others are very difficult. Regression tests over the entire problem database are run every few days. Every month or so, we generate and store the sets of proofs generated by such runs. Differences between runs are noted, especially proofs of theorems that have not been proved before. We go by the principle "if it is too good to be true then it probably is": unexpectedly good results are scrutinised carefully. We have examined many proofs manually. Although it is impractical to check every step of a 60-page proof, questionable sections can be examined, using a text editor to find occurrences of particular terms.

Our methods are based on inequalities, and any formula requiring exactness will probably be impossible to prove. Our bounds are typically exact at a single point. We can exactly bound $\exp x$, $\sin x$, $\cos x$, $\tan x$ and $\tan^{-1} x$ when $x = 0$, and $\ln x$ and $\sqrt{x}$ when $x = 1$. MetiTarski is unlikely to prove any inequality that holds exactly at other points.

To illustrate another issue, consider the following theorem, which is impossible to prove in MetiTarski:

$$|x| < \pi/2 \land x \neq 0 \Longrightarrow 2/\pi < \sin x/x$$

The difficulty is that $\sin x/x = 2/\pi$ when $x = \pi/2$, so if x exceeds $\pi/2$ then the inequality fails. The problem requires the exact value of π, but MetiTarski simply treats π as a variable such that $3.1415926 < \pi < 3.1415927$. Strangely, MetiTarski can prove a slightly weaker version of the theorem:

$$|x| < \pi/2 - 10^{-15} \land x \neq 0 \Longrightarrow 2/\pi < \sin x/x$$

If $p \approx \pi$ to only 7 decimal places, we obviously cannot conclude that $p/2 - 10^{-15} < \pi/2$, and MetiTarski's proof of this theorem seems to suggest a soundness bug. But as p increases, $2/p$ decreases, maintaining $2/p < \sin x/x$ a bit longer. MetiTarski can prove this slight weakening of the problem because π appears twice. Replacing π by two separate variables each confined to the interval (3.1415926, 3.1415927) makes the proof attempt fail.

Another puzzle is that MetiTarski can prove this theorem:

$$|\exp x - 1| \leq \exp|x| - 1$$

It apparently requires finding an algebraic upper bound for $\exp x$ with x arbitrarily large. No such bound exists. Here, however, the absolute value function engenders a case analysis on $x \geq 0$. If true, then also $\exp x - 1 \geq 0$ and the problem reduces to the trivial $\exp x - 1 \leq \exp x - 1$.

The point of these two examples is that we are familiar with our problems and know when to be suspicious of a claimed proof.

5.2 Experimental Design and Detailed Findings

We have compared four combinations of heuristic settings:

- no case splitting
- no backtracking
- backtracking without SOS
- backtracking with an SOS weighting factor of 2.0

We have undertaken runs with time limits from 5 to 300 seconds. Unfortunately, time-limiting is not especially robust. The UNIX command `"ulimit -t s"` allows s seconds to MetiTarski and to its QEPCAD subprocess independently, therefore allowing up to $2s$ seconds. In practice, UNIX seems to allow a few seconds more than s. Nevertheless, time-limiting is applied uniformly in all runs, so we are comparing like with like.

All backtracking runs limit the split stack to 100. These tests were run on a 2.8 GHz Mac Pro workstation, using MetiTarski version 1.9.

Table 1 displays the number of theorems proved in each of the four cases and for a variety of runtimes. Fig. 1 displays these results graphically, in terms of percentages of theorems proved. (The difference between backtracking and backtracking+SOS is too small to see visually, so the graph omits the former.) The clear finding is that any form of splitting is better than none, solving an additional 4–6% of the problem set. Backtracking (with the extra benefit of SOS weighting) solves an additional 1–2% of the problem set compared with non-backtracking. The advantage of backtracking is minimal for short runtimes and increases along with the runtime.

Table 1: Summary of Results: Theorems Proved (out of 791)

	5	15	30	60	120	180	240	300 secs
cases off	517	550	564	584	599	607	609	613
backtracking off	551	585	605	622	632	638	639	642
backtracking	547	585	609	628	638	644	646	649
backtracking + SOS	557	591	616	637	644	651	653	656

5.3 Performance on Hard Problems

A possible criticism of these figures is that they do not distinguish between easy problems, which make up the majority, and hard problems. Fully 65 percent of the problem set can be solved in under five seconds. Let us define a hard problem as one that can be solved in 60 seconds using one heuristic and that cannot be solved in that amount of time using more than two different heuristics. There are 75 such problems. Fig. 2 displays the number of these difficult theorems that can be proved for different combinations of heuristic settings and runtime. For hard problems, the advantages of splitting and backtracking are pronounced, particularly for runtimes below 60 seconds.

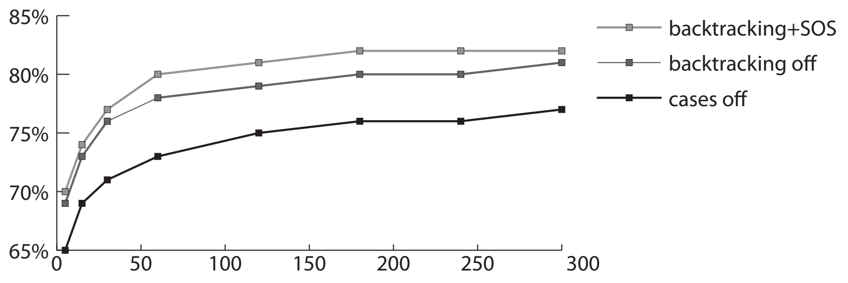

Figure 1: Theorems Proved (by percentage of the total)

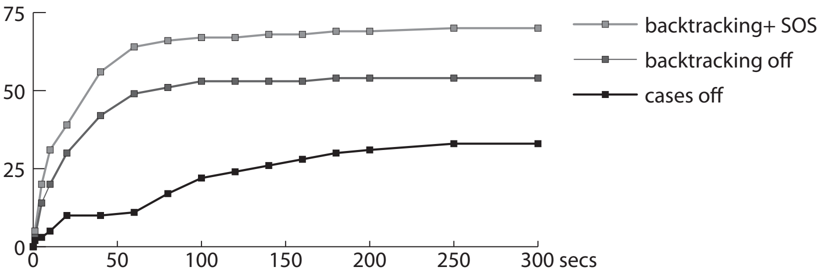

Figure 2: Hard Theorems Proved (out of 75)

Another possible criticism is that there are differences other than backtracking between the two types of splitting. SOS is only used with backtracking; hyper splitting is only used without backtracking. In each case, we have adopted the combination of heuristics leading to the best results. This seems to be the only sensible comparison. If we make the two types of splitting as similar as possible, despite degrading the results, then they will bear no relation to reality. Our comparison is between heuristic settings that are actually available to users.

5.4 Raw Statistics

The question naturally arises, why does splitting with backtracking deliver better results than the other proof methods? In order to investigate this question, we ran a substantial job (allowing 2500 seconds per problem, for 800 problems) collecting the following information:

- total time, Metis time and RCF time in seconds
- given clause count and total number of new clauses
- number of RCF calls, splits and successful subsumptions
- proof size (number of unique clauses)

The above are for each of the four combinations of heuristic settings listed at the beginning of Sect. 5.2. We could not discern many meaningful patterns in this data. For example, if a proof is found quickly, the other statistics tend to be lower than usual, which is unsurprising. Interested readers can download this data for themselves and undertake their own investigations.[2]

One measure is possibly significant: the size of the final proof, in terms of the number of unique clauses. We compared the backtracking and non-backtracking cases for 669 theorems which are proved by both approaches within 300 secs. The median values are similar: 87 with backtracking, 94.5 without backtracking (and 91 for no splitting). Backtracking produced a shorter proof for 210 of the theorems, while non-backtracking did so for 101 theorems, and 358 theorems had proofs of the same length. It appears that backtracking tends to find shorter proofs. But these statistics do not make us much the wiser.

6 Conclusions

Case splitting is a well-known heuristic that has been used since the 1990s. It has grown in popularity and is now available (in some form) in many of the leading automatic theorem provers for first-order logic. We have described the use of case splitting in MetiTarski. Enabled by default, splitting delivers a clear improvement to the success rate. This result is significant, even if (due to MetiTarski's unusual architecture) our results cannot be extrapolated to conventional theorem provers. We restrict attention to ground clauses because virtually all significant MetiTarski problems are effectively ground. (Universally quantified variables in a first-order formula becomes Skolem constants after negation and conversion to clause form.) Non-ground splitting may be necessary in other applications.

We have implemented two versions of splitting: lightweight and backtracking. The latter is considerably harder to implement, requiring pervasive changes to code and data structures. It is superior to lightweight splitting, especially for the most difficult theorems. Moreover, the availability of both methods increases the number of theorems that can be proved in a reasonable amount of processor time. We are not aware of any comparable experiments involving other systems.

[2] `http://www.cl.cam.ac.uk/~lp15/papers/Arith/case-splitting-stats.csv`

Acknowledgements. Joe Hurd answered questions about Metis and commented on this paper, as did Christoph Benzmüller. The research was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/I011005/1 and EP/I010335/1]. The anonymous referees made many insightful comments.

References

[1] Akbarpour, B., Paulson, L.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning **44**(3), 175–205 (2010). DOI 10.1007/s10817-009-9149-2

[2] Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson and Voronkov [12], chap. 2, pp. 19–99

[3] Denman, W., Akbarpour, B., Tahar, S., Zaki, M., Paulson, L.C.: Formal verification of analog designs using MetiTarski. In: A. Biere, C. Pixley (eds.) Formal Methods in Computer Aided Design, pp. 93–100. IEEE (2009). DOI 10.1109/FMCAD.2009.5351136

[4] Fietzke, A., Weidenbach, C.: Labelled splitting. Annals of Mathematics and Artificial Intelligence **55**, 3–34 (2009). DOI 10.1007/s10472-009-9150-9

[5] Hurd, J.: Metis first order prover (2007). URL `http://gilith.com/software/metis/`

[6] McCune, W., Wos, L.: Otter: The CADE-13 competition incarnations. Journal of Automated Reasoning **18**(2), 211–220 (1997). DOI 10.1023/A:1005843632307

[7] Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. Journal of Applied Logic **7**(1), 41–57 (2009). DOI 10.1016/j.jal.2007.07.004

[8] Mitrinović, D.S., Vasić, P.M.: Analytic Inequalities. Springer (1970)

[9] Nivelle, H.: Splitting through new proposition symbols. In: R. Nieuwenhuis, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning: 8th International Conference, LPAR, LNCS 2250, pp. 172–185. Springer (2001). DOI 10.1007/3-540-45653-8_12

[10] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson and Voronkov [12], chap. 6, pp. 335–367

[11] Riazanov, A., Voronkov, A.: Splitting without backtracking. In: International Joint Conference on Artificial intelligence (IJCAI-17) — Volume 1, pp. 611–617 (2001). URL `http://www.cs.man.ac.uk/~voronkov/papers/ijcai01.ps`

[12] Robinson, A., Voronkov, A. (eds.): Handbook of Automated Reasoning. Elsevier Science (2001)

[13] Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: W. Zhang, V. Sorge (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, no. 112 in Frontiers in Artificial Intelligence and Applications, pp. 201–215. IOS Press (2004)

[14] Weidenbach, C.: SPASS - version 0.49. Journal of Automated Reasoning **18**, 247–252 (1997). DOI 10.1023/A:1005812220011

[15] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson and Voronkov [12], chap. 27, pp. 1965–2013

[16] Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM **12**(4), 536–541 (1965). DOI 10.1145/321296.321302. URL `http://doi.acm.org/10.1145/321296.321302`