A bisimulation between DPLL(T) and a proof-search strategy for the focused sequent calculus

Mahfuza Farooque, Stéphane Lengrand, Assia Mahboubi

A Bisimulation between DPLL($\mathcal{T}$) and a Proof-Search Strategy for the Focused Sequent Calculus

Mahfuza Farooque
CNRS - École Polytechnique, France
mahfuza@lix.polytechnique.fr

Stéphane Graham-Lengrand
CNRS - École Polytechnique, France
graham-lengrand@lix.polytechnique.fr

Assia Mahboubi
INRIA, France
assia.mahboubi@inria.fr

Abstract

We describe how the Davis-Putnam-Logemann-Loveland procedure DPLL is bisimilar to the goal-directed proof-search mechanism described by a standard but carefully chosen sequent calculus. We thus relate a procedure described as a transition system on states to the gradual completion of incomplete proof-trees.

For this we use a *focused* sequent calculus for *polarised* classical logic, for which we allow analytic cuts. The focusing mechanisms, together with an appropriate management of polarities, then allows the bisimulation to hold: The class of sequent calculus proofs that are the images of the DPLL runs finishing on UNSAT, is identified with a simple criterion involving polarities.

We actually provide those results for a version DPLL($\mathcal{T}$) of the procedure that is parameterised by a background theory $\mathcal{T}$ for which we can decide whether conjunctions of literals are consistent. This procedure is used for Satisfiability Modulo Theories (SMT) generalising propositional SAT. For this, we extend the standard focused sequent calculus for propositional logic in the same way DPLL($\mathcal{T}$) extends DPLL: with the ability to call the decision procedure for $\mathcal{T}$.

DPLL($\mathcal{T}$) is implemented as a plugin for PSYCHE, a proof-search engine for this sequent calculus, to provide a sequent-calculus based SMT-solver.

*Categories and Subject Descriptors
F.4.1 [*Mathematical Logic*]: Mechanical theorem proving

*Keywords
polarised logic; focused sequent calculus; DPLL($\mathcal{T}$)

1. Introduction

The sequent calculus is a versatile formalism that can be used to describe *goal-directed proof-search*, the foundational paradigm of a broad range of tools, from higher-order proof-assistants to logic programming. Not only is the gradual bottom-up construction of proof-trees in sequent calculus the basis of analytic *tableaux* methods, but it has also been shown to describe mechanisms as diverse as, on the one hand, type inhabitation / proof-construction in (the basic theory of) Coq [24] and, on the other hand, computation in ProLog and λ-ProLog [28].

This paper sets the foundations for applying the same methodology to the automated techniques developed to solve the *Satisfiability Modulo Theories (SMT)* family of problems, making them available to systems based on goal-directed proof-search.

Such problems generalise propositional SAT-problems: instead of considering the satisfiability of conjunctive normal forms (CNF) over propositional variables, SMT problems are concerned with the satisfiability of CNF over atomic propositions from a theory $\mathcal{T}$ such as linear arithmetic or bit vectors. Given a procedure deciding the consistency -with respect to $\mathcal{T}$- of a conjunction of atoms or negated atoms, SMT-solving organises a cooperation between this procedure and SAT-solving techniques, thus providing a decision procedure for SMT-problems. This smart extension of the successful SAT-solving techniques opened a prolific area of research and led to the implementation of ever-improving tools, namely SMT-solvers, now crucial to a number of applications in software verification. The architecture of SMT-solvers is based on the extension of the Davis, Putnam, Logemann and Loveland (DPLL) procedure [13, 14] for solving SAT-problems to a procedure called DPLL($\mathcal{T}$) [32] addressing SMT-problems.

This paper does not try to improve the DPLL($\mathcal{T}$) technique itself, or current SMT-solvers based on it, but makes a step towards the integration of the technique into a sequent calculus framework.

A now wide literature achieves the integration of SMT-solvers in various tools, using the blackbox approach. For instance, several proof assistants propose an infrastructure allowing the user to call an external SMT-solver as a blackbox and re-interpret its output to reconstruct a proof within the system [2, 8, 9, 35]. This blackbox infrastructure is natural, given the available tools, their development history, and the communities that designed their techniques.

Here, we aim at a new and deeper integration where DPLL($\mathcal{T}$) is performed within the system. Recently, an internal implementation of some SMT-techniques was made available in the Coq proof assistant [25], but is very specific to Coq's *reflection* feature [10] (and therefore can hardly be adapted to a framework without reflection).

We rather investigate a broader and more basic context where we can perform each of the *steps* of DPLL($\mathcal{T}$) as the standard steps of proof-search in sequent calculus: the gradual and goal-directed construction of a proof-tree. This allows the DPLL($\mathcal{T}$) algorithm to be applied *up-to-a-point*, where a switch to another technique can be made (depending on the newly generated goals), whereas the use of *reflection* or of a blackbox call only works when the entire goal can be treated by a (full) run of DPLL($\mathcal{T}$).

The results in this paper can be seen as an abstract description of DPLL($\mathcal{T}$) that aims at providing a better proof-theoretical understanding of how different theorem proving techniques (e.g. tableaux, resolution, DPLL($\mathcal{T}$),...), geared towards different logical fragments, could efficiently cooperate inside the same prover.

LFMTP '13, September 23, 2013, Boston, MA, USA.

ACM 978-1-4503-2382-6/13/09.
http://dx.doi.org/10.1145/2503887.2503892

This was not explicitly the concern of previous abstract descriptions of DPLL($\mathcal{T}$) that have been proposed in the literature for the purpose of studying the non-trivial properties of its implementations in proof-theoretical terms. These use resolution trees [7] or most often transition systems [31, 32] based on rewrite rules.

While offering a seemingly convenient way of representing branching (and hence backtracking), the sequent calculus turns out to be a somewhat more rigid setting (than the aforementioned transition systems), tied to the root-first decomposition of formula-trees. Other descriptions of DPLL($\mathcal{T}$) based on trees (e.g. [34]) offer some work-arounds, but are specifically designed for the job (of describing DPLL($\mathcal{T}$)).

In this paper we show how proof-search in a rather standard sequent calculus, called $\mathsf{LK}^p(\mathcal{T})$, can simulate DPLL($\mathcal{T}$). More precisely, we identify an *elementary* version of DPLL($\mathcal{T}$) that is the direct extension of the *Classical DPLL procedure* to a background theory $\mathcal{T}$, as well as being a restriction of the *Full DPLL Modulo Theories* system,[1] both of which can be found in [32].

For this we do not tailor the sequent calculus to DPLL($\mathcal{T}$) but we do use well-known sequent calculus features: not only do we allow the use of *analytic cuts*, but we also use *polarities* and *focusing*. Arising from Linear Logic [1, 17], the last two features also make sense in classical logic [18, 22, 26]: while Gentzen's original rules offer a lot of non-determinism in the proof-search, these features provide a tight control on the breadth of the search space.[2]

Such control allows us to derive a stronger result than the mere simulation of DPLL($\mathcal{T}$): the proofs in $\mathsf{LK}^p(\mathcal{T})$ that are the images of DPLL($\mathcal{T}$) runs finishing on UNSAT can be characterised by a simple criterion only involving the way polarities are assigned to literals and the way formulae are placed into the *focus* of sequents (the device implementing focusing). From this criterion we directly get a simple proof-search strategy that is bisimilar to DPLL($\mathcal{T}$) runs: that which performs the depth-first completion of incomplete proof-trees (starting with the leftmost open leaf), using any inference steps satisfying the given criterion on polarities and focusing.

That way, we ensure that bottom-up proof-search in sequent calculus can be as efficient as the DPLL($\mathcal{T}$) procedure.

In order to validate the theoretical simulation of DPLL($\mathcal{T}$), and to evaluate its efficiency as a proof-search method in $\mathsf{LK}^p(\mathcal{T})$, we have implemented the simulation as a plugin for PSYCHE [20, 33], a proof-search tool based on $\mathsf{LK}^p(\mathcal{T})$. This tool has a modular architecture in a style similar to LCF [29]: a simple kernel offers proof-search primitives for $\mathsf{LK}^p(\mathcal{T})$, and plugins are programmed with these primitives to drive the kernel through the construction of a proof-tree. Thanks to this modularity, the correctness of PSYCHE's output only relies on that of the simple kernel, while efficiency of proof-search is left to the plugin that implements an identified proof-search strategy of interest. We discuss the performance of our DPLL($\mathcal{T}$) plugin on a standard benchmark.

The paper is organised as follows: section 2 describes the sequent calculus $\mathsf{LK}^p(\mathcal{T})$ and section 3 presents the elementary system for DPLL($\mathcal{T}$); section 4 presents the simulation of that system in $\mathsf{LK}^p(\mathcal{T})$; section 5 identifies the corresponding proof-search strategy in $\mathsf{LK}^p(\mathcal{T})$ to complete the bisimulation. Finally, section 6 provides an overview of the PSYCHE proof-search tool and of its DPLL($\mathcal{T}$) plugin, before we discuss related works and perspectives in sections 7 and 8.

[1] that allows more advanced features such as *backjumping*

[2] In brief, the inference rules decomposing the connectives of the same polarity can be chained without loosing completeness - see e.g. [1, 28].

2. Focusing for proof-search

In this section we briefly review the proof-search motivation for focused proof systems, and present the focused sequent calculus $\mathsf{LK}^p(\mathcal{T})$ for propositional classical logic *modulo a theory*.

2.1 Background

At the basis of logic programming, proof-search on *Horn clauses* can be understood as a meaningful computational paradigm because this class of formulae makes a simple goal-directed proof-search strategy logically complete (with well-identified backtrack points and a reasonably efficient covering of the proof-search space). This still holds when the class is extended to *hereditary Harrop formulae* [28], and can hold on a wider class of formulae if logical connectives and atoms are tagged with *polarities*: positive or negative. Polarities emerged with the help of *linear logic* [17] and Andreoli's *focusing* results [1], informally described below:

- Negative connectives can be decomposed, in sequent calculus style, with *invertible* inference rules that are called *asynchronous*: a proof-search strategy can perform the bottom-up application of those rules as basic proof-search steps without loss of generality (if the goal was provable, it remains provable after applying the step); in other words, no backtracking is necessary on the application of such steps, even though other steps were possible.
- Positive connectives are the (De Morgan's) duals of negative connectives, and their decomposition rules, which are called *synchronous*, are not necessarily invertible.

Clearly, asynchronous rules can be applied eagerly, i.e. can be chained, without creating backtrack points and losing completeness; it turns out that synchronous rules (although possibly creating backtrack points) can also be chained without losing completeness. This result can be expressed as the completeness of a sequent calculus with a *focus* device, which syntactically highlights a formula in the sequent and forces the next proof-search step to decompose it with a synchronous rule, keeping the focus on its newly-revealed sub-formulae. Focusing considerably reduces the proof-search space, otherwise heavily redundant when Gentzen-style inference rules are used.

A sequent with a positive atom in focus must be proved immediately by an axiom on that atom; hence, the polarity of atoms greatly affects the shape of proofs. As illustrated in e.g. [26], the following sequent expresses the *Fibonacci* logic program (in some language where addition is primitive) and a goal $\mathsf{fib}(n, p)$ (where n and p are closed terms):

$$\mathsf{fib}(0,0),\ \mathsf{fib}(1,1),\ \forall i p_1 p_2(\mathsf{fib}(i,p_1) \Rightarrow \mathsf{fib}(i+1,p_2) \Rightarrow \mathsf{fib}(i+2,p_1+p_2)) \vdash \mathsf{fib}(n,p)$$

The goal will be proved with backward-reasoning if the fib atoms are negative (yielding a proof of exponential size in n), and forward-reasoning if they are positive (yielding many proofs, one of which being linear).

In *classical* logic, polarities of connectives and atoms do not affect the provability of formulae, but still greatly affect the shape of proofs, and hence the basic proof-construction steps. This paper shows how the DPLL($\mathcal{T}$) steps correspond to proof-construction steps for an appropriate management of polarities. Our focused sequent calculus $\mathsf{LK}^p(\mathcal{T})$ for classical logic builds on previous systems based on LC [12, 18, 22], and is syntactically closest to LKF [26], now quite standard.

In order to make logical sense of e.g. the primitive addition in the Fibonacci example above, we only enrich LKF with the ability to call a decision procedure to decide the consistency of conjunctions of literals w.r.t. a theory (i.e. the same as for DPLL($\mathcal{T}$)): for

a theory that equates $1 + 1$ and 2, a call to the procedure proves $p(2), p^{\perp}(1 + 1) \vdash$ in one step (unlike LKF's syntactic checks).

System LKF also assumes that all atoms come with a predetermined polarity, whereas $\mathsf{LK}^p(\mathcal{T})$ allows *on-the-fly* polarisation of atoms: the root of a proof-tree might have none of its atoms polarised, but atoms may become positive or negative as progress is made in the proof-search.

2.2 The $\mathsf{LK}^p(\mathcal{T})$ sequent calculus with analytic cuts

This sequent calculus (and this logic) involves a notion of *literal* and a notion of *theory*. The reader can safely see behind this terminology the standard notions from proof theory and automated reasoning. However at this point, very little is required from or assumed about those two notions.

Definition 1 (Literals)
Let $\mathcal{L}$ be a set of elements called *literals*, equipped with an involutive function called *negation* from $\mathcal{L}$ to $\mathcal{L}$. In the rest of this paper, a possibly primed or indexed lowercase l always denotes a literal, and $l^{\perp}$ its negation.

Another ingredient of $\mathsf{LK}^p(\mathcal{T})$ is a *theory* $\mathcal{T}$, given in the form of an *inconsistency predicate*, a notion that we now introduce:

Definition 2 (Inconsistency predicates)
An *inconsistency predicate* is a predicate over sets of literals
- satisfied by the set $\{l, l^{\perp}\}$ for every literal l;
- that is upward closed (if a subset of a set satisfies the predicate, so does the set);
- such that if the sets $\mathcal{P}, l$ and $\mathcal{P}, l^{\perp}$ satisfy it then so does $\mathcal{P}$.

The smallest inconsistency predicate is called the *syntactical inconsistency* predicate[3]. If a set $\mathcal{P}$ of literals satisfies the syntactically inconsistency predicate, we say that $\mathcal{P}$ is *syntactically inconsistent*, denoted $\mathcal{P} \models$. Otherwise $\mathcal{P}$ is *syntactically consistent*.

The theory $\mathcal{T}$ in the notation $\mathsf{LK}^p(\mathcal{T})$ is described by means of an(other) inconsistency predicate, called the *semantical inconsistency predicate*, which will be a formal parameter of the inference system defining $\mathsf{LK}^p(\mathcal{T})$.

If a set $\mathcal{P}$ of literals satisfies the semantical inconsistency predicate, we say that $\mathcal{P}$ is *semantically inconsistent* or *inconsistent modulo theory*, denoted by $\mathcal{P} \models_{\mathcal{T}}$. Otherwise $\mathcal{P}$ is *semantically consistent* or *consistent modulo theory*.

Definition 3 (Formulae, negation)
Let $\mathcal{L}$ be a set of literals. The formulae of propositional polarised classical logic are given by the following grammar:

$$\begin{array}{lrcll} \text{Formulae} & A, B, \ldots & ::= & l & \text{where } l \text{ ranges over } \mathcal{L} \\ & & | & A \wedge^{+} B \mid A \vee^{+} B \mid \top^{+} \mid \perp^{+} & \\ & & | & A \wedge^{-} B \mid A \vee^{-} B \mid \top^{-} \mid \perp^{-} & \end{array}$$

The size of a formula A, denoted $\sharp(A)$, is its size as a tree (number of nodes).

Let $\mathcal{P} \subseteq \mathcal{L}$ be syntactically consistent.
Intuitively, it represents the set of literals declared to be positive.

We define $\mathcal{P}$-positive formulae and $\mathcal{P}$-negative formulae as the formulae generated by the following grammars:

$$\begin{array}{ll} \mathcal{P}\text{-positive formulae} & P, \ldots ::= p \mid A \wedge^{+} B \mid A \vee^{+} B \mid \top^{+} \mid \perp^{+} \\ \mathcal{P}\text{-negative formulae} & N, \ldots ::= p^{\perp} \mid A \wedge^{-} B \mid A \vee^{-} B \mid \top^{-} \mid \perp^{-} \end{array}$$

where p ranges over $\mathcal{P}$.
Formulae that are neither $\mathcal{P}$-positive nor $\mathcal{P}$-negative are said to be $\mathcal{P}$-unpolarised.[4]

[3] It is the predicate that is true of a set $\mathcal{P}$ of literals iff $\mathcal{P}$ contains both l and $l^{\perp}$ for some $l \in \mathcal{L}$.

[4] These are necessarily literals.

Negation is recursively extended into an involutive map on formulae as follows:

$(A \wedge^{+} B)^{\perp}$	$:= A^{\perp} \vee^{-} B^{\perp}$	$(A \wedge^{-} B)^{\perp}$	$:= A^{\perp} \vee^{+} B^{\perp}$
$(A \vee^{+} B)^{\perp}$	$:= A^{\perp} \wedge^{-} B^{\perp}$	$(A \vee^{-} B)^{\perp}$	$:= A^{\perp} \wedge^{+} B^{\perp}$
$(\top^{+})^{\perp}$	$:= \perp^{-}$	$(\top^{-})^{\perp}$	$:= \perp^{+}$
$(\perp^{+})^{\perp}$	$:= \top^{-}$	$(\perp^{-})^{\perp}$	$:= \top^{+}$

Remark 1 Note that, given a syntactically consistent set $\mathcal{P}$ of literals, negations of $\mathcal{P}$-positive formulae are $\mathcal{P}$-negative and vice versa.

Notation 4 A possibly primed or indexed Γ always denotes a set of formulae. By Γ_{lit} we denote the subset of elements of Γ that are literals, and we write $l \in \Gamma$ if l or $l^{\perp}$ appears in Γ.

Definition 5 (System $\mathsf{LK}^p(\mathcal{T})$)
The system $\mathsf{LK}^p(\mathcal{T})$ is the sequent calculus defined by the rules of Figure 1, which fall into three categories: *synchronous*, *asynchronous*, and *structural* rules, and manipulate two kinds of sequents:

$\Gamma \vdash^{\mathcal{P}} [A]$ where the formula A is in the *focus* of the sequent
$\Gamma \vdash^{\mathcal{P}} \Gamma'$

where $\mathcal{P}$ is a syntactically consistent set of literals declared to be positive.
A sequent of the second kind where Γ' is empty is called *developed*.

The gradual proof-tree construction defined by the bottom-up application of the inference rules of $\mathsf{LK}^p(\mathcal{T})$, is a goal-directed mechanism whose intuition can be given as follows:

Asynchronous rules are invertible: $(\wedge^{-})$ and $(\vee^{-})$ are applied eagerly when trying to construct the proof-tree of a given sequent; (Store) is applied when hitting a positive formula or a negative literal on the right-hand side of a sequent, storing its negation on the left; (Pol) is the *on-the-fly* polarisation rule, applied on demand, for instance when a right-hand side literal is of undetermined polarity and therefore cannot yet be stored.

When the right-hand side of a sequent becomes empty (i.e. the sequent is *developed*), a sanity check can be made with (Init_2) to check the semantical consistency of the stored literals (w.r.t. the theory), otherwise a choice must be made to place a positive formula in focus, using rule (Select), before applying synchronous rules like $(\wedge^{+})$ and $(\vee^{+})$. Each such rule decomposes the formula in focus, keeping the revealed sub-formulae in the focus of the corresponding premises, until a positive literal or a non-positive formula is obtained: the former case must be closed immediately with (Init_1) calling the decision procedure, and the latter case uses the (Release) rule to drop the focus and start applying asynchronous rules again. The synchronous and the structural rules are in general not invertible,[5] so each application of those yields in general a backtrack point in the proof-search.

Notice that an invariant of such a proof-tree construction process is that the left-hand side of a sequent only contains negative formulae and positive literals.

Notation 6 When F is a formula of unpolarised propositional logic and Ψ is a set of such formulae, $\Psi \models F$ means that Ψ entails F in propositional classical logic. Given a theory $\mathcal{T}$ (given by a semantical inconsistency predicate), we define the set of all *theory lemmas* as $\Psi_{\mathcal{T}} := \{l_1 \vee \cdots \vee l_n \mid l_1^{\perp}, \cdots, l_n^{\perp} \models_{\mathcal{T}}\}$ and generalise the notation $\models_{\mathcal{T}}$ to write $\Psi \models_{\mathcal{T}} F$ when $\Psi_{\mathcal{T}}, \Psi \models F$. In that case we say that F is a *semantical consequence* of Ψ. For any polarised

[5] (but they may be so, e.g. $(\wedge^{+})$)

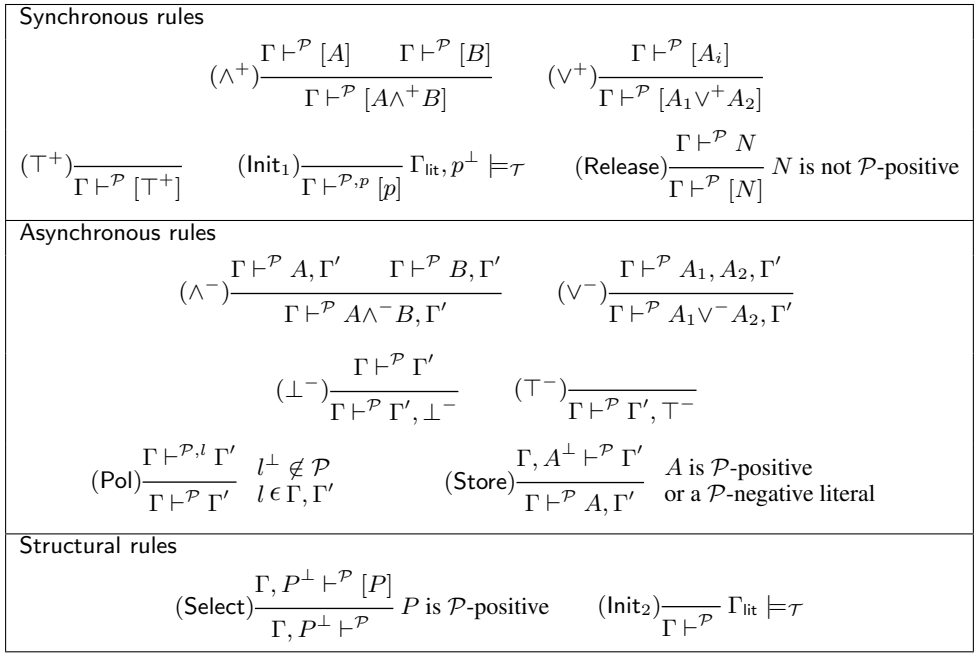

Figure 1. System $\mathsf{LK}^p(\mathcal{T})$

formula A, let $\underline{A}$ be the unpolarised formula obtained by removing all polarities on connectives.

Theorem 2 (Cut-elimination and Completeness of $\mathsf{LK}^p(\mathcal{T})$ [16])

- *The following* analytic cut-rule *is admissible in $\mathsf{LK}^p(\mathcal{T})$:*
$$(\mathsf{cut})\frac{\Gamma \vdash^{\mathcal{P}} l^\perp \quad \Gamma \vdash^{\mathcal{P}} l}{\Gamma \vdash^{\mathcal{P}}}\ l \in \Gamma$$
- *If $\models_{\mathcal{T}} F$, then for all A such that $\underline{A} = F$ and all $\mathcal{P}$, we can prove $\vdash^{\mathcal{P}} A$ in $\mathsf{LK}^p(\mathcal{T})$.*

The meta-theory of $\mathsf{LK}^p(\mathcal{T})$, in particular the proofs of the above, can be found in [16].

3. The elementary DPLL($\mathcal{T}$) procedure

Intuitively, $\mathsf{DPLL}(\mathcal{T})$ aims at proving the inconsistency of a set of *clauses* with respect to a theory. We therefore retain from the previous section the notion of literal and inconsistencies, and introduce clauses:

Definition 7 (Clause)
A *clause* is a finite set of literals, which can be seen as their disjunction.

In the rest of the paper, a possibly indexed upper cased C always denotes a clause. The empty clause is denoted by $\perp$. The number of literals in a clause C is denoted $\sharp(C)$. The possibly indexed symbol ϕ always denotes finite sets of clauses $\{C_1, \ldots, C_n\}$, which can also be seen as a Conjunctive Normal Form (CNF). We use $\sharp(\phi)$ to denote the sum of the sizes of the clauses in ϕ. Finally $\mathsf{lit}(\phi)$ denotes the set of literals that appear in ϕ or whose negations appear in ϕ.

Viewing clauses as disjunctions of literals and sets of clauses as CNF, we will generalise Notation 6, writing for instance $\phi \models \neg C$ or $\phi \models C$, as well as $\phi \models_{\mathcal{T}} \neg C$ or $\phi \models_{\mathcal{T}} C$.

Definition 8 (Decision literals and sequences)
We consider a (single) copy of the set $\mathcal{L}$ of literals, denoted $\mathcal{L}^d$, whose elements are called *decision literals*, which are just tagged clones of the literals in $\mathcal{L}$. Decision literals are denoted[6] by l^d.

We use the possibly indexed symbol Δ to denote a finite sequence of possibly tagged literals, with $\emptyset$ denoting the empty sequence. We also use Δ_1, Δ_2 and Δ_1, l, Δ_2 to denote the suggested concatenation of sequences.

For such a sequence Δ, we write $\overline{\Delta}$ for the subset of $\mathcal{L}$ containing all the literals in Δ with their potential tags removed. The sequences that $\mathsf{DPLL}(\mathcal{T})$ will construct will always be duplicate-free, so the difference between Δ and $\overline{\Delta}$ is just a matter of tags and ordering. When the context is unambiguous, we will sometimes use Δ when we mean $\overline{\Delta}$.

We define $\mathsf{Clo}(\Delta) := \{l \mid \Delta, l^\perp \models_{\mathcal{T}}\}$, the closure of a sequence Δ by semantical entailment. For any set of clauses ϕ, the set of literals occuring in ϕ that are semantically entailed by Δ is denoted by $\mathsf{Clo}_\phi(\Delta) := \mathsf{Clo}(\Delta) \cap \mathsf{lit}(\phi)$.

Remark 3 Semantical consequences are the analogues of the consequences of a partial boolean assignment in the context of a DPLL procedure for propositional logic without theory. Obviously, if $l \in \Delta$, then $l \in \mathsf{Clo}(\Delta)$. If $\phi_1 \subseteq \phi_2$, then for any Δ, $\mathsf{Clo}_{\phi_1}(\Delta) \subseteq \mathsf{Clo}_{\phi_2}(\Delta)$.

We can now describe the elementary $\mathsf{DPLL}(\mathcal{T})$ procedure as a transition system between states.

Definition 9 (Elementary DPLL($\mathcal{T}$))
A *state* of the $\mathsf{DPLL}(\mathcal{T})$ procedure is either the state UNSAT, or a pair denoted $\Delta \| \phi$, where ϕ is a set of clauses and Δ is a sequence of possibly tagged literals. The *transition rules* of the elementary $\mathsf{DPLL}(\mathcal{T})$ procedure are given in Fig. 2.

[6] This exponent tag is a standard notation, standing for "decision".

Decide	$\Delta\|\phi$	$\Rightarrow \Delta, l^d\|\phi$	where $l \in \mathsf{lit}(\phi)$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Propagate	$\Delta\|\phi, C \vee l$	$\Rightarrow \Delta, l\|\phi, C \vee l$	where $\Delta \models \neg C$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Propagate$_\mathcal{T}$	$\Delta\|\phi$	$\Rightarrow \Delta, l\|\phi$	where $l \in \mathsf{Clo}_\phi(\Delta)$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Fail	$\Delta\|\phi, C$	$\Rightarrow$ UNSAT,	where $\Delta \models \neg C$	and there is no decision literal in Δ.
Fail$_\mathcal{T}$	$\Delta\|\phi$	$\Rightarrow$ UNSAT,	where $\Delta \models_\mathcal{T}$	and there is no decision literal in Δ.
Backtrack	$\Delta_1, l^d, \Delta_2\|\phi, C$	$\Rightarrow \Delta_1, l^\perp\|\phi, C$	where $\Delta_1, l, \Delta_2 \models \neg C$	and there is no decision literal in Δ_2.
Backtrack$_\mathcal{T}$	$\Delta_1, l^d, \Delta_2\|\phi$	$\Rightarrow \Delta_1, l^\perp\|\phi$	where $\Delta_1, l, \Delta_2 \models_\mathcal{T}$	and there is no decision literal in Δ_2.

Figure 2. Elementary DPLL($\mathcal{T}$)

This transition system is an extension of the *Classical DPLL procedure*, as presented in [32], to the background theory $\mathcal{T}$.[7] The first four rules are explicitly taken from the Abstract DPLL Modulo Theories system of [32].[8] The other rules of that system (namely $\mathcal{T}$-Backjump, $\mathcal{T}$-Learn, $\mathcal{T}$-Forget, etc), are not considered here in their full generality, but specific cases and combinations are covered by the rest of our elementary DPLL($\mathcal{T}$) system, so that it is logically complete.[9] Note that this transition system is not deterministic: for instance the Decide rule can be applied from any state and it furthermore does not enforce a strategy for picking the literal to be tagged among the eligible elements of $\mathsf{lit}(\phi)$. At the level of implementation, this (non deterministic) transition system is turned into a deterministic algorithm, whose efficientcy crucially relies on the strategies adopted to perform the choices left unspecified by DPLL($\mathcal{T}$).

We illustrate those rules, in the theory $\mathcal{T}$ of *Linear Rational Arithmetic*, with the two basic examples of elementary DPLL($\mathcal{T}$) runs presented in Fig. 3 (where Δ and ϕ always refer to the current state $\Delta\|\phi$).

A reason to introduce rule Fail$_\mathcal{T}$ is to allow the second run to finish with the same output as the first: Indeed, the last Propagate step has created a $\mathcal{T}$-inconsistency from which we could not derive UNSAT without a Fail$_\mathcal{T}$ step.[10]

4. Simulation of the elementary DPLL($\mathcal{T}$) procedure in LK^p($\mathcal{T}$)

The aim of this section is to describe how the elementary DPLL($\mathcal{T}$) procedure can be transposed into a proof-search process for sequents of the LK^p($\mathcal{T}$) calculus. A complete and successful run of the DPLL($\mathcal{T}$) procedure is a sequence of transitions $\emptyset\|\phi \Rightarrow^*$ UNSAT, which ensures that the set of clauses ϕ is inconsistent modulo the theory. Hence, we are devising a proof-search process aiming at building an LK^p($\mathcal{T}$) proof-tree for sequents of the form $\phi' \vdash$, where ϕ' represents the set of clauses ϕ as a sequent calculus structure, in the following sense:

Definition 10 (Representation of clauses as formulae)
An LK^p($\mathcal{T}$) formula C' *represents* a DPLL($\mathcal{T}$) clause $\{l_j\}_{j=1\ldots p}$ if $C' = l_1 \vee^- \ldots \vee^- l_p \vee^- \perp^-$.

A set of formulae ϕ' *represents* a set of clauses ϕ if there is a bijection f from ϕ to ϕ' such that for all clauses C in ϕ, $f(\phi)$ represents C.

[7] We removed the Pure Literal rule, in general unsound in presence of a theory $\mathcal{T}$.

[8] Unit Propagate and Theory Propagate are renamed as Propagate and Propagate$_\mathcal{T}$ for consistency with the other rule names.

[9] Backtrack is a restricted version of $\mathcal{T}$-Backjump (this holds on the basis that the full system satisfies some basic invariant -Lemma 3.6 of [32]), Fail$_\mathcal{T}$ (resp. Backtrack$_\mathcal{T}$) is a combination of $\mathcal{T}$-Learn, Fail (resp. Backtrack), and $\mathcal{T}$-Forget steps.

[10] (or, alternatively, a $\mathcal{T}$-Learn step in [32])

Remark 4 If C' represents C, then $\sharp(C') \leq 2\sharp(C)$ (there are fewer symbols $\vee^-$ than there are literals in C).

Note here that we carefully use the negative disjunction connective to translate DPLL($\mathcal{T}$) clauses. This is crucial not only to mimic DPLL($\mathcal{T}$) without duplicating formulae but more generally to control the search space.

Now, in order to construct a proof of $\phi' \vdash$ from a run $\emptyset\|\phi \Rightarrow^*$ UNSAT, we proceed gradually by considering the intermediate steps of the DPLL($\mathcal{T}$) run:

$$\emptyset\|\phi \Rightarrow^* \Delta\|\phi \Rightarrow^* \text{UNSAT}$$

In the intermediate DPLL($\mathcal{T}$) state $\Delta\|\phi$, the sequence Δ is a log of both the search space explored so far (in $\emptyset\|\phi \Rightarrow^* \Delta\|\phi$) and the search space that remains to be explored (in $\Delta\|\phi \Rightarrow^*$ UNSAT). In this log, a tagged decision literal l^d indicates a point where the procedure has made an exploratory choice (the case where l is true has been/is being explored, the case where $l^\perp$ is true remains to be explored), while untagged literals in Δ are predictable consequences of the decisions made so far and of the set of clauses ϕ to be falsified.

If we are to express the DPLL($\mathcal{T}$) procedure as the gradual construction of a LK^p($\mathcal{T}$) proof-tree, we should get from $\emptyset\|\phi \Rightarrow^* \Delta\|\phi$ a proof-tree that is not yet complete and get from $\Delta\|\phi \Rightarrow^*$ UNSAT some (complete) proof-tree(s) that can be "plugged into the holes" of the incomplete tree. We should read in Δ the "interface" between the incomplete tree that has been constructed and the complete sub-trees to be constructed.

We use the plural here since there can be more than one subtree left to construct: $\Delta\|\phi \Rightarrow^*$ UNSAT contains the information to build not only a proof of $\overline{\Delta}, \phi' \vdash$, but also proofs of the sequents corresponding to the other parts of the search space to be explored, characterised by the tagged literals in Δ. For instance, a run from $l_1, l_2^d, l_3, l_4^d\|\phi \Rightarrow^*$ UNSAT contains the information to build a proof of $l_1, l_2, l_3, l_4, \phi' \vdash$ but also the proofs of $l_1, l_2, l_3, l_4^\perp, \phi' \vdash$ and $l_1, l_2^\perp, \phi' \vdash$. Those extra sequents are obtained by collecting from a sequence Δ its "backtrack points" as follows:

Definition 11 (Backtrack points)
The *backtrack points* $[\![\Delta]\!]$ of a sequence Δ of possibly tagged literals is the list of sets of untagged literals recursively defined by the following rules, where $[\,]$ and :: are the standard list constructors.

$$\begin{array}{lcl} [\![()]\!] & := & [\,] \\ [\![\Delta, l]\!] & := & \underline{[\![\Delta]\!]} \\ [\![\Delta, l^d]\!] & := & \overline{\Delta}, l^\perp :: [\![\Delta]\!] \end{array}$$

Remark 5 The length of $[\![\Delta]\!]$ is the number of decision literals in Δ.

Now, coming back to the DPLL($\mathcal{T}$) transition sequence $\emptyset\|\phi \Rightarrow^* \Delta\|\phi$ and its intuitive counterpart in sequent calculus, we have to formalise the notion of *incomplete* proof-tree together with the notion of "filling its holes":

$$
\begin{array}{r||ll}
\emptyset & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate} \\
x>0 & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate} \\
x>0,(x+y>0)^{\bot} & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate}_{\mathcal{T}} \quad ((y>0)^{\bot}\in \mathsf{Clo}_{\phi}(\Delta)) \\
x>0,(x+y>0)^{\bot},(y>0)^{\bot} & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate}_{\mathcal{T}} \quad ((x=-1)^{\bot}\in \mathsf{Clo}_{\phi}(\Delta)) \\
x>0,(x+y>0)^{\bot},(y>0)^{\bot},(x=-1)^{\bot} & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Fail} \quad \text{on clause } (y>0\vee x=-1) \\
\mathsf{UNSAT} &&
\end{array}
$$

$$
\begin{array}{r||ll}
\emptyset & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate} \\
x>0 & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate} \\
x>0,(x+y>0)^{\bot} & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate}_{\mathcal{T}} \quad ((y>0)^{\bot}\in \mathsf{Clo}_{\phi}(\Delta)) \\
x>0,(x+y>0)^{\bot},(y>0)^{\bot} & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Propagate}\ (x=-1) \text{ from } (y>0\vee x=-1) \\
x>0,(x+y>0)^{\bot},(y>0)^{\bot},(x=-1) & x>0,(x+y>0)^{\bot},(y>0\vee x=-1) & \mathsf{Fail}_{\mathcal{T}} \quad x>0,x=-1 \text{ inconsistent with } \mathcal{T} \\
\mathsf{UNSAT} &&
\end{array}
$$

Figure 3. Examples of elementary DPLL($\mathcal{T}$) runs

Definition 12 (Incomplete proof-tree, extension)
An *incomplete proof-tree* in $\mathsf{LK}^p(\mathcal{T})$ is a tree labelled with sequents,

- whose leaves are tagged as either *open* or *closed*;
- whose open leaves are labelled with developed sequents;
- and such that every node that is not an open leaf, together with its children, forms an instance of the $\mathsf{LK}^p(\mathcal{T})$ rules.

The *size* of an incomplete proof-tree is its number of nodes.

An incomplete proof-tree π' is an *extension* of π, if there is a tree (edge and nodes preserving) homomorphism from π to π'. It is an n*-extension* of π, if moreover the difference of size between π' and π is less than or equal to n.

Remark 6 An incomplete proof-tree that has no open leaf is (isomorphic to) a well-formed complete $\mathsf{LK}^p(\mathcal{T})$ proof of the sequent labelling its root. In that case, we say the proof-tree is *complete*.

The intuition that an intermediate DPLL($\mathcal{T}$) state describes an "interface" between an incomplete proof-tree and the complete proof-trees that should be plugged into its holes, is formalised as follows:

Definition 13 (Correspondance)
An incomplete proof-tree π *corresponds to* a DPLL($\mathcal{T}$) state $\Delta\|\phi$ if:

- the length of $\overline{\Delta} :: [\![\Delta]\!]$ is the number of open leaves of π;
- if Δ_i is the i^{th} element of $\overline{\Delta} :: [\![\Delta]\!]$, then the i^{th} open leaf of π (taken left-to-right) is labelled by a developed sequent of the form $\Delta'_i, \phi'_i \vdash^{\Delta_i}$, where:
  - ϕ'_i represents ϕ (in the sense of Definition 10);
  - $\mathsf{Clo}_{\phi}(\Delta_i) = \mathsf{Clo}_{\phi}(\Delta'_i)$.

An incomplete proof-tree π *corresponds to* the state UNSAT if it has no open leaf.

Remark 7 In the general case, different incomplete proof-trees might correspond to the same DPLL($\mathcal{T}$) state (just like different DPLL($\mathcal{T}$) runs may reach that state from the initial one).

Note that we do not require anything from the conclusion of an incomplete proof-tree corresponding to $\Delta\|\phi$: just as our correspondence says nothing about the DPLL($\mathcal{T}$) transitions taking place after $\Delta\|\phi$ (nor about the trees to be plugged into the open leaves), it says nothing about the transitions taking place before $\Delta\|\phi$ (nor about the incomplete proof-tree, except for its open leaves).

If an incomplete proof-tree π corresponds to a DPLL($\mathcal{T}$) state $\Delta\|\phi$ where there are no decision literals in Δ, then there is exactly one open leaf in π, and it is labelled by a sequent of the form $\Delta', \phi' \vdash^{\overline{\Delta}}$, where ϕ' represents ϕ and $\mathsf{Clo}_{\phi}(\Delta) = \mathsf{Clo}_{\phi}(\Delta')$.

To the initial state $\emptyset\|\phi$ of a run of the DPLL($\mathcal{T}$) procedure corresponds the incomplete proof-tree consisting of one node (both root and open leaf) labelled with the sequent $\phi' \vdash$, where ϕ' represents ϕ.

The simulation theorem below provides a systematic way of interpreting any DPLL($\mathcal{T}$) transition as a completion of incomplete proof-trees that preserves the correspondence given in Definition 13 and controls the growth of the proof trees.

Theorem 8 (Simulation of DPLL($\mathcal{T}$) in $\mathsf{LK}^p(\mathcal{T})$)
If $\Delta\|\phi \Rightarrow \mathcal{S}_2$ is a valid DPLL($\mathcal{T}$) transition, and π_1 is an incomplete proof tree in $\mathsf{LK}^p(\mathcal{T})$ corresponding to $\Delta\|\phi$, then there exists a $(2\sharp(\phi)+3)$-extension π_2 of π_1 that corresponds to $\mathcal{S}_2$.

Proof: By case analysis on the nature of the transition, completing the leftmost open leaf of π_1:

- Decide:
  $\Delta\|\phi \Rightarrow \Delta, l^d\|\phi$ where $l \notin \Delta, l^{\bot} \notin \Delta, l \in \mathsf{lit}(\phi)$.
  Let π_1 be an incomplete proof-tree corresponding to $\Delta\|\phi$. The leftmost leaf (corresponding to $\overline{\Delta}$) is of the form $\Delta', \phi' \vdash^{\overline{\Delta}}$ where ϕ' represents ϕ and $\mathsf{Clo}_{\phi}(\overline{\Delta}) = \mathsf{Clo}_{\phi}(\Delta')$.
  We extend π_1 into π_2 by replacing the leftmost leaf by the following (incomplete) proof-tree:
  $$\dfrac{\dfrac{\dfrac{\Delta', l, \phi' \vdash^{\overline{\Delta},l}}{\Delta', \phi' \vdash^{\overline{\Delta},l}\ l^{\bot}}}{\Delta', \phi' \vdash^{\overline{\Delta}} l^{\bot}} \qquad \dfrac{\dfrac{\Delta', l^{\bot}, \phi' \vdash^{\overline{\Delta},l^{\bot}}}{\Delta', \phi' \vdash^{\overline{\Delta},l^{\bot}}\ l}}{\Delta', \phi' \vdash^{\overline{\Delta}} l}}{\Delta', \phi' \vdash^{\overline{\Delta}}}$$
  Note that we use here the analytic cut rule of $\mathsf{LK}^p(\mathcal{T})$. π_2 is a 3-extension of π_1 that corresponds to $\Delta, l^d\|\phi$. Indeed, we have $\overline{\Delta, l^d} :: [\![\Delta, l^d]\!] = (\overline{\Delta}, l) :: (\overline{\Delta}, l^{\bot}) :: [\![\Delta]\!]$ and $\mathsf{Clo}_{\phi}(\overline{\Delta}, l) = \mathsf{Clo}_{\phi}(\Delta', l)$ and $\mathsf{Clo}_{\phi}(\overline{\Delta}, l^{\bot}) = \mathsf{Clo}_{\phi}(\Delta', l^{\bot})$. The two new leaves are tagged as open.
- Propagate:
  $\Delta\|\phi, C \vee l \Rightarrow \Delta, l\|\phi, C \vee l$ where $\Delta \models \neg C, l \notin \Delta, l^{\bot} \notin \Delta$.
  Let π_1 be an incomplete proof-tree corresponding to $\Delta\|\phi, C \vee l$. The open leaf corresponding to $\overline{\Delta}$ is of the form $\Delta', \phi', C' \vdash^{\overline{\Delta}}$ where ϕ' represents ϕ, C' represents $C \vee l$ and $\mathsf{Clo}_{\phi, C\vee l}(\overline{\Delta}) =$

$\mathsf{Clo}_{\phi,C\vee l}(\Delta')$. Let $C = l_1 \vee \ldots \vee l_n$. From $\Delta \models \neg C$ we get $\forall i, l_i^{\perp} \in \overline{\Delta} \subseteq \mathsf{Clo}_{\phi,C\vee l}(\overline{\Delta}) = \mathsf{Clo}_{\phi,C\vee l}(\Delta')$.

We extend π_1 into π_2 by replacing this leaf by the following (incomplete) proof-tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta', l, \phi', C' \vdash^{\overline{\Delta},l}}{\Delta', \phi', C' \vdash^{\overline{\Delta},l} l^{\perp}}}{\Delta', \phi', C' \vdash^{\overline{\Delta}} l^{\perp}}}{\Delta', \phi', C' \vdash^{\overline{\Delta}} [l^{\perp}]} \quad \left(\dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta}} [l_i^{\perp}]} \right)_{l_i \in C} \quad \dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta}} [\top^{+}]}}{\vdots (\wedge^{+}).}$$

$$\dfrac{\Delta', \phi', C' \vdash^{\overline{\Delta}} [C'^{\perp}]}{\Delta', \phi', C' \vdash^{\overline{\Delta}}}$$

The top-right rules can be applied since $l_i^{\perp} \in \overline{\Delta}$ and $\Delta', l_i \models_{\mathcal{T}}$ and the new leaves are closed. The top-left leaf is tagged as open.

Noticing that $\overline{\Delta, l} :: [\![\Delta, l]\!] = (\overline{\Delta}, l) :: [\![\Delta]\!]$, we get that π_2 is a $\sharp(C') + 3$-extension of π_1 that corresponds to $\Delta, l \| \phi, C \vee l$ (and note that $\sharp(C') \leq 2\sharp(\phi)$).

- $\mathsf{Propagate}_{\mathcal{T}}$:
  $\Delta \| \phi \Rightarrow \Delta, l \| \phi$ where $l \in \mathsf{Clo}_{\phi}(\Delta)$ and $l \notin \Delta, l^{\perp} \notin \Delta$.

  Let π_1 be an incomplete proof-tree corresponding to $\Delta \| \phi$. The open leaf corresponding to $\overline{\Delta}$ is of the form $\Delta', \phi' \vdash^{\overline{\Delta}}$ where ϕ' represents ϕ and $\mathsf{Clo}_{\phi}(\Delta) = \mathsf{Clo}_{\phi}(\Delta')$. We extend π_1 into π_2 by replacing this leaf by the following (incomplete) proof-tree:

  $$\dfrac{\Delta', \phi' \vdash^{\overline{\Delta},l}}{\Delta', \phi' \vdash^{\overline{\Delta}}}$$

  Noticing that $\mathsf{Clo}_{\phi}(\Delta) = \mathsf{Clo}_{\phi}(\Delta, l)$, π_2 is a 1-extension of π_1 that corresponds to $\Delta, l \| \phi$.

- Fail:
  $\Delta \| \phi, C \Rightarrow \mathsf{UNSAT}$ where $\Delta \models \neg C$ and there is no decision literal in Δ.

  Let π_1 be an incomplete proof-tree corresponding to $\Delta \| \phi, C$. Since there are no decision literals in Δ, π_1 has exactly one open leaf, and it is labelled by $\Delta', \phi', C' \vdash^{\overline{\Delta}}$ where ϕ' represents ϕ, C' represents C and $\mathsf{Clo}_{\phi,C}(\Delta) = \mathsf{Clo}_{\phi,C}(\Delta')$. Let $C = l_1 \vee \ldots \vee l_n$.

  From $\Delta \models \neg C$ we get $\forall i,\ l_i^{\perp} \in \overline{\Delta} \subseteq \mathsf{Clo}_{\phi,C}(\Delta) = \mathsf{Clo}_{\phi,C}(\Delta')$.

  We extend π_1 into π_2 by replacing the open leaf by the following (complete) proof-tree:

  $$\dfrac{\left(\dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta}} [l_i^{\perp}]} \right)_{l_i \in C} \quad \dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta}} [\top^{+}]}}{\vdots \wedge^{+}.}$$

  $$\dfrac{\Delta', \phi', C' \vdash^{\overline{\Delta}} [C'^{\perp}]}{\Delta', \phi', C' \vdash^{\overline{\Delta}}}$$

  The top rules can be applied since $l_i^{\perp} \in \overline{\Delta}$ and $\Delta', l_i \models_{\mathcal{T}}$. All the leaves are closed. π_2 is a $\sharp(C') + 1$-extension of π_1 that is complete, and therefore corresponds to the UNSAT state of the $\mathsf{DPLL}(\mathcal{T})$ run (and note that $\sharp(C') \leq 2\sharp(\phi)$).

- $\mathsf{Fail}_{\mathcal{T}}$:
  $\Delta \| \phi \Rightarrow \mathsf{UNSAT}$ where $\Delta \models_{\mathcal{T}}$ and there is no decision literal in Δ.

  Let π_1 be an incomplete proof-tree corresponding to $\Delta \| \phi$. Since there are no decision literals in Δ, π_1 has exactly one open leaf, and it is labelled by $\Delta', \phi' \vdash^{\overline{\Delta}}$ where ϕ' represents ϕ and $\mathsf{Clo}_{\phi}(\Delta) = \mathsf{Clo}_{\phi}(\Delta')$.

  We extend π_1 into π_2 by replacing the open leaf by the following (complete) proof-tree:

  $$\dfrac{}{\Delta', \phi' \vdash^{\overline{\Delta}}} \Delta', \phi' \models_{\mathcal{T}}$$

  Here, π_2 is a 1-extension of π_1 that is complete and corresponds to UNSAT state of the $\mathsf{DPLL}(\mathcal{T})$ run.

- $\mathsf{Backtrack}$:
  $\Delta_1, l^d, \Delta_2 \| \phi, C \Rightarrow \Delta_1, l^{\perp} \| \phi, C$ where $\Delta_1, l, \Delta_2 \models \neg C$ and no decision literal is in Δ_2.

  Let π_1 be an incomplete proof-tree corresponding to $\Delta_1, l^d, \Delta_2 \| \phi, C$. The open leaf corresponding to $\overline{\Delta_1, l^d, \Delta_2}$ is of the form $\Delta', \phi', C' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}}$ where ϕ' represents ϕ, C' represents C and $\mathsf{Clo}_{\phi}(\Delta_1, l, \Delta_2) = \mathsf{Clo}_{\phi}(\Delta')$. Let $C = l_1 \vee \ldots \vee l_n$.

  From $\overline{\Delta_1}, l, \overline{\Delta_2} \models \neg C$ we get $\forall i,\ l_i^{\perp} \in \overline{\Delta_1}, l, \overline{\Delta_2} \subseteq \mathsf{Clo}_{\phi,C}(\overline{\Delta_1}, l, \overline{\Delta_2}) = \mathsf{Clo}_{\phi,C}(\Delta')$.

  We extend π_1 into π_2 by replacing this leaf by the following (complete) proof-tree:

  $$\dfrac{\left(\dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}} [l_i^{\perp}]} \right)_{l_i \in C} \quad \dfrac{}{\Delta', \phi', C' \vdash^{\overline{\Delta},l,\overline{\Delta_2}} [\top^{+}]}}{\vdots \wedge^{+}.}$$

  $$\dfrac{\Delta', \phi', C' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}} [C'^{\perp}]}{\Delta', \phi', C' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}}}$$

  The top rules can be applied since $l_i^{\perp} \in \overline{\Delta_1}, l, \overline{\Delta_2}$ and $\Delta', l_i \models_{\mathcal{T}}$.

  Noticing that $[\![\Delta_1, l^d, \Delta_2]\!] = (\overline{\Delta_1}, l^{\perp}) :: [\![\Delta_1]\!]$, we get that π_2 is a $\sharp(C') + 1$-extension of π_1 that corresponds to $\Delta_1, l^{\perp} \| \phi, C$ (and note that $\sharp(C') \leq 2\sharp(\phi)$).

- $\mathsf{Backtrack}_{\mathcal{T}}$:
  $\Delta_1, l^d, \Delta_2 \| \phi \Rightarrow \Delta_1, l^{\perp} \| \phi$ if $\Delta_1, l, \Delta_2 \models_{\mathcal{T}}$ and no decision literal is in Δ_2.

  Let π_1 be an incomplete proof-tree corresponding to $\Delta_1, l^d, \Delta_2 \| \phi$. The open leaf corresponding to $\overline{\Delta_1, l^d, \Delta_2}$ is of the form $\Delta', \phi' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}}$ where ϕ' represents ϕ and $\mathsf{Clo}_{\phi}(\Delta_1, l, \Delta_2) = \mathsf{Clo}_{\phi}(\Delta')$.

  We extend π_1 into π_2 by replacing this leaf by the following (complete) proof-tree:

  $$\dfrac{}{\Delta', \phi' \vdash^{\overline{\Delta_1},l,\overline{\Delta_2}}} \Delta', \phi' \models_{\mathcal{T}}$$

  Noticing that $[\![\Delta_1, l^d, \Delta_2]\!] = (\overline{\Delta_1}, l^{\perp}) :: [\![\Delta_1]\!]$, we get that π_2 is a 1-extension of π_1 that corresponds to $\Delta_1, l^{\perp} \| \phi$ state of the $\mathsf{DPLL}(\mathcal{T})$ run.

□

Corollary 9

If $\emptyset \| \phi \Rightarrow^n \mathit{UNSAT}$ and ϕ' represents ϕ then there is a complete proof in $\mathsf{LK}^p(\mathcal{T})$ of $\phi' \vdash$, of size smaller than $(2\sharp(\phi) + 3)n$.

5. Completing the bisimulation

Now the point of having mentioned quantitative information in Theorem 8, via the notion of n-extension, is to motivate the idea that performing proof-search directly in $\mathsf{LK}^p(\mathcal{T})$ is in essence not less efficient than running $\mathsf{DPLL}(\mathcal{T})$: we have a linear bound in the length of the $\mathsf{DPLL}(\mathcal{T})$ run (and the proportionality ratio is itself an affine function of the size of the original problem).

We also need to make sure that this final proof-tree is indeed found as efficiently as running $\mathsf{DPLL}(\mathcal{T})$, which can be done by identifying, in $\mathsf{LK}^p(\mathcal{T})$, a (complete) search space that is isomorphic to (and hence no wider than) that of $\mathsf{DPLL}(\mathcal{T})$. We analyse for this a proof-search strategy, in $\mathsf{LK}^p(\mathcal{T})$, that exactly captures the proof-extensions that we have used in the simulation of $\mathsf{DPLL}(\mathcal{T})$, i.e. the proof of Theorem 8:

Definition 14 ($\mathsf{DPLL}(\mathcal{T})$-extensions)

An incomplete proof tree π_2 is a *$\mathsf{DPLL}(\mathcal{T})$-extension* of an incomplete proof tree π_1 if

1. it extends π_1 by replacing its leftmost open leaf with an incomplete proof-tree of one of the forms:

$$\dfrac{\dfrac{\cdots}{\Gamma, A^\perp \vdash^{\mathcal{P}} [A]}\,(b)}{\Gamma, A^\perp \vdash^{\mathcal{P}}}\,(a) \qquad \dfrac{\Gamma \vdash^{\mathcal{P}} l \quad \Gamma \vdash^{\mathcal{P}} l^\perp}{\Gamma \vdash^{\mathcal{P}}}\, l \in \Gamma$$

$$\dfrac{\Gamma \vdash^{\mathcal{P}, l}}{\Gamma \vdash^{\mathcal{P}}}\,(c) \qquad \dfrac{}{\Gamma \vdash^{\mathcal{P}}}\, \Gamma_{\mathsf{lit}} \models_{\mathcal{T}}$$

   where
   (a) A is a (positive) conjunction of literals that are all in $\mathcal{P}$ except maybe one that is $\mathcal{P}$-unpolarised
   (b) the only instances of (Pol) in the above proof are of the form $\dfrac{\Gamma \vdash^{\mathcal{P}, l^\perp} l}{\Gamma \vdash^{\mathcal{P}} l}$
   (c) $l \in \Gamma$ with $\Gamma_{\mathsf{lit}}, l^\perp \models_{\mathcal{T}}$
2. any incomplete proof-tree satisfying point 1. and extended by π_2 is π_2 itself.

Given a $\mathsf{DPLL}(\mathcal{T})$-extension, we can now identify a $\mathsf{DPLL}(\mathcal{T})$ transition that the extension simulates, in the sense of Theorem 8:

Theorem 10 (Simulation of the strategy back into $\mathsf{DPLL}(\mathcal{T})$)

If π_2 is a $\mathsf{DPLL}(\mathcal{T})$-extension of π_1, and π_1 corresponds to $\Delta \| \phi$, then there is a (unique) $\mathsf{DPLL}(\mathcal{T})$ transition $\Delta \| \phi \Rightarrow \mathcal{S}_2$ such that π_2 corresponds to $\mathcal{S}_2$.

Proof: By case analysis on the shape of the incomplete proof-tree replacing the leftmost open leaf of π_1. Out of the four shapes of definition 14:

- for the first one: if there is no $\mathcal{P}$-unpolarised literal in A, it is simulated by a Fail or $\mathsf{Backtrack}$ (depending on whether π_2 is complete) on the clause represented by $A^\perp$; if not, it is simulated by $\mathsf{Propagate}$ on the clause represented by $A^\perp$;
- the second one is simulated by Decide on l;
- the third one is simulated by $\mathsf{Propagate}_{\mathcal{T}}$ on l.
- the fourth one is simulated by $\mathsf{Fail}_{\mathcal{T}}$ or $\mathsf{Backtrack}_{\mathcal{T}}$ (depending on whether π_2 is complete).

The details are the same as in the proof of Theorem 8. □

If a complete proof-tree of $\mathsf{LK}^p(\mathcal{T})$, whose conclusion is an SMT-problem,[11] systematically uses the rules in the way described by the above shapes, then it is the image of a $\mathsf{DPLL}(\mathcal{T})$ run.

While it could be envisaged to simulate $\mathsf{DPLL}(\mathcal{T})$ in a Gentzen-style sequent calculus (with a variant of Theorem 8), the above definition and theorem reveal the advantage of using a focused sequent calculus for polarised logic: Definition 14 presents[12] different ways of *starting* the extension of an open branch (whose leaf sequent is developed), each one of them corresponding to a specific $\mathsf{DPLL}(\mathcal{T})$ transition; then *focusing* takes care of the following steps of the extension so that, when hitting developed sequents again, the exact simulation of the $\mathsf{DPLL}(\mathcal{T})$ transition has been performed.

In order for proof-search mechanisms to exactly match $\mathsf{DPLL}(\mathcal{T})$ transitions, focusing therefore provides the right level of granularity and (together with an appropriate management of polarities) the right level of determinism.

Corollary 11 (Bisimulation)

The correspondance relation (see definition 13) between incomplete proof trees and $\mathsf{DPLL}(\mathcal{T})$ states is a bisimulation for the transition system defined on incomplete proof-trees of $\mathsf{LK}^p(\mathcal{T})$ by the strategy of $\mathsf{DPLL}(\mathcal{T})$-extensions and on states by $\mathsf{DPLL}(\mathcal{T})$.

Finally, obtaining this tight result is the reason why we identified the *elementary $\mathsf{DPLL}(\mathcal{T})$* system, a restriction of the Abstract DPLL Modulo Theories system of [32]:

Modern SMT-solvers feature some mechanisms that are not part of our (logically complete) *elementary $\mathsf{DPLL}(\mathcal{T})$* system but increase efficiency, such as *backjumping* and *lemma learning* (cf. rules $\mathcal{T}$-Backjump, $\mathcal{T}$-Learn in [32]).

It is possible to simulate those rules in $\mathsf{LK}^p(\mathcal{T})$ by using general cuts, by extending with identical steps several open branches of incomplete proof-trees, and possibly by using explicit weakenings (depending on whether we adapt the correspondence between $\mathsf{DPLL}(\mathcal{T})$ states and incomplete proof-trees).

But the price of this is high at the theoretical level: With such "parallel extensions" of incomplete proof-trees, it is not clear how to count the *sizes* of proofs and extensions in a meaningful way, so the quantitative aspects of Theorem 8 and Corollary 9 are compromised; neither is it clear which criterion on proof-trees (and on how to extend them) identifies the proof-construction strategy that is the exact image of a $\mathsf{DPLL}(\mathcal{T})$ procedure featuring those advanced mechanisms. In other words, it is not clear how to obtain such a tight correspondence.

6. $\mathsf{DPLL}(\mathcal{T})$ in PSYCHE

The above description of $\mathsf{DPLL}(\mathcal{T})$ as a proof-search mechanism of $\mathsf{LK}^p(\mathcal{T})$ has been implemented as a plugin for PSYCHE [20, 33], a proof-search engine for $\mathsf{LK}^p(\mathcal{T})$.

6.1 PSYCHE's overview

PSYCHE is a proof-search tool for the sequent calculus presented in Section 2, implemented in OCaml. The construction of a proof-tree for a given formula results from the interaction between a small kernel and plugins, which only agree on the data structures used in their interaction. The former is parametrised by a decision procedure for $\mathcal{T}$ and implements the rules in $\mathsf{LK}^p(\mathcal{T})$ (bottom-up), offering an API to apply synchronous rules (mainly, the choice of focus) while automatically applying asynchronous rules which are invertible, and therefore represent no backtrack point. A plugin implements a strategy that drives the kernel, by calling its API

[11] i.e. it corresponds to an initial state of $\mathsf{DPLL}(\mathcal{T})$

[12] mostly by specifying the management of polarities

functions, towards either a proof or the guarantee that no proof exists. Plugins can be used to import efficient automated reasoning techniques in this goal-directed framework by specifying in which order (and to which depth) the branches of the search-space should be explored. The implementation of plugins is however not trusted for soundness: the worst that a plugin can do is crash the program, not affect its output. A expanded description of PSYCHE can be found in [20].

6.2 A plugin for DPLL($\mathcal{T}$)

Version 1.5 of PSYCHE comes with a plugin implementing the simulation of DPLL($\mathcal{T}$) as described in Section 4. Every time a rule is applied, the plugin calls the API function that takes as input a formula to focus on, feeding it with the appropriate clause. That function also accepts the alternative instruction of making a cut, which is what the plugin uses to simulate Decide. Backtrack and Propagate are done eagerly by the technique of watched literals [30].

As mentioned in the previous section, the theoretical simulation of the full DPLL($\mathcal{T}$) system, with *backjumping* and *lemma learning* requires extending several branches of open proof-trees with parallel steps. To avoid compromising on efficiency, our plugin for PSYCHE implements these advanced features, without departing from the theory described in this paper but using an alternative technique: it simply uses *memoisation* for the proof-search function. This is used to close, in one single step, any branch that would otherwise be closed by repeating the same steps as in another subproof. In particular, doing this avoids repeating, several times, the proof-construction steps of a "parallel extension" corresponding to a single backjump.

Memoisation is also a way of performing clause-learning: Even though our DPLL($\mathcal{T}$) plugin never actually adds a learnt clause to the original set of clauses (which it could actualy do with a general cut), it rather relies on the memoisation table: a learnt clause C is a clause for which we know that $\phi \models_{\mathcal{T}} C$, and that is made available for Fail, Backtrack or Propagate. Such a clause corresponds to a key $\phi', C^{\perp} \vdash$ of the memoisation table, with its proof as value. A state where C can be used for Fail or Backtrack is necessarily a sequent weakening $\phi', C^{\perp} \vdash$ with extra formulae or literals, so the proof recorded in the memoisation table can be plugged there to close the current branch. When C can be used for Propagate, it suffices to make a cut on the missing literal: one branch will be closed by plugging-in the proof recorded in the memoisation table, while the other branch will continue the simulation.

The memoisation table is filled-in by clause-learning: our plugin adds an entry whenever it builds a complete proof of some sequent $\Delta \vdash$ and no previous entry $\Delta' \vdash$ exists with $\Delta' \subseteq \Delta$, or whenever it concludes that some sequent $\Delta \vdash$ is not provable and no previous entry $\Delta' \vdash$ exists with $\Delta \subseteq \Delta'$. For the table to cut computation as often as possible, a pre-processing step is applied to a proof-tree before it enters the table: it is pruned from every formula that is not used in the proof, which is easy to do for complete proofs (eager weakening is applied a posteriori by inspection of the inductive structure). PSYCHE's kernel instead performs pruning *on-the-fly*, whenever an inference is added to complete proofs. Since proof-completion can be seen as finding a conflict, pruning by eager weakening is a conflict analysis process naturally provided by structural proof theory. Of course, the efficiency of pruning relies on the efficiency of the decision procedure in providing a small inconsistent subset whenever it decides that a set of literals is inconsistent.

6.3 Examples

PSYCHE's webpage (examples section) shows the output of PSYCHE when it is run, with its DPLL($\mathcal{T}$) plugin, on the instance that we used in Fig. 3 to illustrate the elementary DPLL($\mathcal{T}$) system. On such a small example, the LaTeX source produced by PSYCHE can be compiled, producing in pdf format the proof-tree in $\mathsf{LK}^p(\mathcal{T})$ (that corresponds to the second run of Fig. 3).

Treating bigger problems, we have evaluated the efficiency of PSYCHE on two standard benchmarks for automated decision procedures. The first set of problems is composed of purely propositional formulae (SAT problems) and is a subset of the Satisfiability Library (SATLIB) benchmark [21]. The second set of problems is composed of quantifier-free problems in the theory of linear rational arithmetic (QFLRA) and is a fragment of the Satisfiability Modulo Theory Library (SMTLIB) benchmark [3]. All the results can be found on the PSYCHE website [33]. Comparing the current implementation of PSYCHE with state-of-the-art SAT and SMT solvers would make little sense: the current version of PSYCHE plugin does not incorporate techniques that are now folklore for SAT solvers, nor any kind of pre-processing. Moreover the decision procedure used by version 1.5 of PSYCHE for linear arithmetic is a very naive version of the simplex algorithm, and most importantly it does not implement the incremental version of this algorithm which is best suited to the needs of an SMT solver. We however believe that the performance results obtained by PSYCHE are promising and we plan to enrich its collection of plugins in order to benefit at least from pre-processing [15] and restart policies [19]. We also plan to interface PSYCHE with decision procedures for a broader range of theories, in order to be able to cover more benchmarks from the SMTLIB.

7. Related Work

In this paper we used the focused sequent calculus $\mathsf{LK}^p(\mathcal{T})$ for polarised classical logic, which considerably narrows the search space provided by Gentzen's sequent calculus. As already mentioned in Section 2.1, $\mathsf{LK}^p(\mathcal{T})$ is a variant of LKF [26], enriched with the ability to polarise atoms *on-the-fly*, and of course the ability to call a decision procedure as in DPLL($\mathcal{T}$).

We also allow an *analytic cut-rule*, notwithstanding that proof-search in sequent calculus is usually done in a cut-free system, as the cut-rule usually unreasonably widens the search-space. Analytic cuts only concern atomic cut-formulae among the finitely many atoms present in the rest of the sequent. Allowing them does widen the search space (which we narrowed in other ways) but this sometimes permits to draw quicker conclusions, in a way similar to DPLL($\mathcal{T}$)'s Decide rule.

Miller and Nigam [27] have already shown how to use analytic cuts to incorporate *tables* into proofs, i.e. make sure that, once an atom is proved or known to be true (from a table of lemmas), the subsequent proof-search never tries to re-prove it. This is achieved by giving, to the atom that is cut, two opposite polarities in the two premisses of the cut. The simulation of DPLL($\mathcal{T}$)'s Decide rule in $\mathsf{LK}^p(\mathcal{T})$ uses the same trick.

Moreover, their approach seems to have strong links with the memoisation table that our PSYCHE implementation uses to avoid reproving sequents. So far, our memoisation table is not reflected in the sequent calculus itself, but we could envisage adapting their approach, possibly capturing the interaction with the memoisation table by the use of general cuts. Proof-search may then depart from the mere gradual construction of a proof-tree. But an appealing idea is that the cleverness that goes into finding a Theory lemma (i.e. a key of the memoisation table) translates as the cleverness that goes into picking a good cut-formula during proof-search. Indeed, for the mere simulation of our elementary DPLL($\mathcal{T}$) system, no memoisation table is required, just as no cut-formula ever needs to be picked.

Also note that the simulation of elementary DPLL($\mathcal{T}$) is tight and can be quantified: the bounds that we computed show that

proof-search in $\mathsf{LK}^p(\mathcal{T})$ is no less efficient. Lifting this to the advanced features would require taking memoisation and/or general cuts into account (future work). This also hints at the field of proof complexity: the power of DPLL with or without backjumping, clause learning, etc, has been connected to proof complexity via Resolution systems [7]. But there is also a literature on proof complexity in sequent calculus (with/without cuts), which our approach should relate to.

While resolution systems have also been a way to relate DPLL and its variants to formal proof theory, the present paper was motivated by the import of such algorithms in a goal-directed framework, on which many systems (e.g. ProLog) are based. Resolution trees have also been used to perform conflict analysis; but since abstract presentations of $\mathsf{DPLL}(\mathcal{T})$ [32] abstract away the conflict analysis method (so as to accommodate any strategy), so did we (in the theory). On the other hand, our implementation in PSYCHE performs a conflict analysis according to a proof-theoretical method: eager weakenings a posteriori; and this we intend to compare to conflict analysis methods based on resolution trees or conflict graphs.

Finally, a similar interaction between generic reasoning mechanisms and domain-specific methods can be found in the field of Constraint Logic Programming (CLP), but usually using traditional fragments of logic such as that of Horn clauses or Hereditary Harrop Formulae [23]. In a sense, system $\mathsf{LK}^p(\mathcal{T})$ could be seen as going one step further to full (polarised) classical logic, making it a natural candidate framework to relate SMT and CLP techniques. However, the handling of quantifiers is an important aspect of CLP, so relating our work to [23] would require extending $\mathsf{LK}^p(\mathcal{T})$ with quantifiers, which is work-in-progress.

8. Conclusion and Further work

In this paper we have identified an elementary $\mathsf{DPLL}(\mathcal{T})$ procedure and established a bisimulation with the gradual construction of proof-trees in $\mathsf{LK}^p(\mathcal{T})$ according to a simple strategy.

While $\mathsf{LK}^p(\mathcal{T})$ differs from the inference system of e.g. [34] (e.g. we still take formulae to be trees and inference rules to organise the root-first decomposition of their connectives, rather than using $\mathsf{DPLL}(\mathcal{T})$'s more flexible structures), it would be interesting to capture some of the related systems that extend $\mathsf{DPLL}(\mathcal{T})$ with e.g. full first-order logic and/or equality [4–6]. The full version of $\mathsf{LK}^p(\mathcal{T})$ is indeed designed for handling quantifiers and equalities, so we hope to relate it to other techniques such as unification, paramodulation, superposition, etc.

This is also our challenge for the PSYCHE implementation: ideally, try to show how the smart mechanisms that we know to be efficient at proving something, can be emulated or decomposed into an interaction between our kernel and a specific plugin to be programmed. This would turn PSYCHE into a modular and collaborative platform, a standard format of problem-solving mechanisms where each technique can play its part, be it from start to finish, or in collaboration with each other or a human user.

We can imagine using a proof assistant to prove PSYCHE's correctness [11], since the kernel is small, not using any imperative features, and the plugins need not be certified. This approach is an alternative to existing techniques of certification of SMT tools (or of any other technique that could instead be implemented as a PSYCHE plugin).

Acknowledgments

This work has benefitted from the ANR-09-JCJC-0006-01 grant from the *Agence Nationale de le Recherche*. The authors are very grateful to the anonymous referees for their constructive comments, and in particular the reference to [23].

References

[1] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic Comput.*, 2(3):297–347, 1992.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proc. of the 1st Int. Conf. on Certified Programs and Proofs (CPP'11)*, volume 7086 of *LNCS*, pages 135–150. Springer, Dec. 2011. ISBN 978-3-642-25378-2.

[3] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2010. `www.SMT-LIB.org`.

[4] P. Baumgartner. FDPLL - a first order Davis-Putnam-Longeman-Loveland procedure. In *Proc. of the 17th Int. Conf. on Automated Deduction (CADE'00)*, volume 1831 of *LNCS*, pages 200–219. Springer-Verlag, June 2000. ISBN 3-540-67664-3.

[5] P. Baumgartner and C. Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172(4-5):591–632, 2008.

[6] P. Baumgartner and C. Tinelli. Model evolution with equality modulo built-in theories. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proc. of the 23rd Int. Conf. on Automated Deduction (CADE'11)*, volume 6803 of *LNCS*, pages 85–100. Springer-Verlag, July 2011. ISBN 978-3-642-22437-9.

[7] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artificial Intelligence Res.*, 22:319–351, 2004.

[8] F. Besson, P.-E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 151–166. Springer-Verlag, 2011. ISBN 978-3-642-25378-2.

[9] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *Automated Deduction*, volume 6803 of *LNCS*, pages 116–130. Springer-Verlag, 2011.

[10] S. Boutin. Using reflection to build efficient and certified decision procedure s. In M. Abadi and T. Ito, editors, *TACS'97*, volume 1281 of *LNCS*. Springer-Verlag, 1997.

[11] A. Charguéraud. Program verification through characteristic formulae. In P. Hudak and S. Weirich, editors, *Proc. of the 15th ACM Intern. Conf. on Functional Programming*, pages 321–332. ACM Press, 2010. `http://arthur.chargueraud.org/research/2010/cfml`.

[12] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: Linear logic. *J. of Symbolic Logic*, 62(3):755–807, 1997.

[13] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of the ACM Press*, 7(3):201–215, 1960.

[14] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[15] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Proc. of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 61–75. Springer-Verlag, 2005.

[16] M. Farooque and S. Graham-Lengrand. Sequent calculi with procedure calls. Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, Parsifal - INRIA Saclay, France, Jan. 2013. `http://hal.archives-ouvertes.fr/hal-00779199`.

[17] J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–101, 1987.

[18] J.-Y. Girard. A new constructive logic: Classical logic. *Math. Structures in Comput. Sci.*, 1(3):255–296, 1991.

[19] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In J. Mostow and C. Rich, editors, *Proc. of the 15th National Conf. on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conf. (IAAI'98)*, pages 431–437. The MIT press, 1998.

[20] S. Graham-Lengrand. Psyche: a proof-search engine based on sequent calculus with an LCF-style architecture. In D. Galmiche and D. Larchey-Wendling, editors, *Proc. of the 22nd Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'13)*, volume 8123 of *LNCS*. Springer-Verlag, Sept. 2013.

[21] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the year 2000*, Frontiers Artificial Intelligence Appl., pages 283–292. Kluwer Academic Publishers, 2000.

[22] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, Mar. 2002.

[23] J. Leach, S. Nieva, and M. Rodríguez-Artalejo. Constraint logic programming with hereditary harrop formula. *Theory and Practice of Logic Programming*, 1(4):409–445, 2001.

[24] S. Lengrand, R. Dyckhoff, and J. McKinna. A focused sequent calculus framework for proof search in Pure Type Systems. *Logic. Methods Comput. Science*, 7(1), 2011.

[25] S. Lescuyer and S. Conchon. Improving Coq propositional reasoning using a lazy CNF conversion scheme. In *Proc. of the 7th Int. Conf. on Frontiers of combining systems (FroCoS'09)*, pages 287–303. Springer-Verlag, 2009. ISBN 3-642-04221-X, 978-3-642-04221-8. `http://dl.acm.org/citation.cfm?id=1807707.1807727`.

[26] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoret. Comput. Sci.*, 410(46):4747–4768, 2009. `http://dx.doi.org/10.1016/j.tcs.2009.07.041`.

[27] D. Miller and V. Nigam. Incorporating tables into proofs. In *Proc. of the 16th Annual Conf. of the European Association for Computer Science Logic (CSL'07)*, volume 4646 of *LNCS*, pages 466–480. Springer-Verlag, 2007. ISBN 978-3-540-74914-1.

[28] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51: 125–157, 1991.

[29] R. Milner. LCF: A way of doing proofs with a machine. In J. Becvár, editor, *Proc. of the the 8th Int. Symp. on Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 146–159. Springer-Verlag, 1979.

[30] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of DAC'01*, pages 530–535, New York, NY, USA, 2001. ACM. `http://doi.acm.org/10.1145/378239.379017`.

[31] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL Modulo Theories. In F. Baader and A. Voronkov, editors, *Proc. of the the 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *LNCS*, pages 36–50. Springer-Verlag, Mar. 2005. ISBN 3-540-25236-3.

[32] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. of the ACM Press*, 53(6):937–977, 2006.

[33] Psyche: the Proof-Search factorY for Collaborative HEuristics. `http://www.lix.polytechnique.fr/~lengrand/Psyche`.

[34] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. of the 8th European Conf. on Logics in Artificial Intelligence*, volume 2424 of *LNAI*, pages 308–319. Springer-Verlag, 2002.

[35] T. Weber. SMT solvers: New oracles for the HOL theorem prover. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):419–429, 2011. `http://dx.doi.org/10.1007/s10009-011-0188-8`.