Master research Internship

Master Thesis

Formalisation of Ground Inference Systems in a Proof Assistant

Domain: Data Structures and Algorithms - Logic in Computer Science

Author:
Mathias Fleury

Supervisor:
Jasmin Blanchette
Dmitriy Traytel
Automation of Logic
(Max-Planck-Institut für Informatik,
Saarbrücken)

Abstract

Various methods have been developed for solving SAT problems, notably resolution, the Davis-Putnam-Logemann-Loveland-Procedure procedure (DPLL) and an extension of it, the conflict-driven clause learning (CDCL). We have formalised these three algorithms in a proof assistant Isabelle/HOL, based on a chapter of Christoph Weidenbach's upcoming book *Automed Reasoning – The Art of Generic Problem Solving*. The three calculi are presented uniformly as transition systems. We have formally proved that each calculus is a decision procedure for satisfiability of propositional logic. One outcome of the formalization is a verified SAT solver based on DPLL and implemented in a functional programming language.

Contents

1 Introduction

Automated reasoning has been successfully applied to find bugs and prove the absence of bugs in various systems. Robinson has developed modern resolution in 1965 [1]. It works on classical first order logic. The basic idea is to use “modus ponens” on a set of clauses and compute all the consequences of the clauses and see if $\bot$ (falsity) is among all the consequences. If falsity is present, then the set of clauses is not satisfiable. Resolution has been latter extended to allow the use of the equality predicate, because it is tedious to axiomatize and inefficient to only axiomatize the equality. The search space can be reduced via term ordering: we still use “modus ponens”, but there are more side-conditions to reduce the number of clauses that can be deduced. The completeness of this method can be proven with a sophisticated proof: when all consequences have been calculated, there is no model if and only if falsity has not been deduced. The method is the base of successful provers, notably E [2], SPASS [3] and Vampire [4].

Another branch of automated reasoning is based on satisfiability solving: given a boolean formula, is there an assignment of the atoms such that the formula is true? The first procedure is the Davis-Putnam-Logemann-Loveland (DPLL) procedure: the basic idea is to split. The procedure is a bit like a truth table: it performs case distinctions on the propositional variables, which can take the value $\top$ (truth) or $\bot$ (falsity). The advantage is that the clauses get simpler, but we have to backtrack on the choices: as in a truth table, each literal is either true or false and both cases have to be tested. The DPLL procedure has been extended to conflict-driven clause learning (CDCL) that does less backtracking thanks by learning from the conflicts (the formula that are false). The satisfiability modulo theories (SMT) is the extension of the previous procedures to first order logic and other theories (e.g. arithmetic). It the basis of successful provers, notably CVC4 [5] and Z3 [6].

Resolution on one side and DPLL and CDCL on the other side have different strength and weaknesses: resolution has a better handling on quantifiers, while SMT solvers handle large ground problems better. Automated theorem provers (ATPs) have a rich metatheory (for example there is a semi-decision procedure for first order logic), but most of it has been developed only on paper. Another community, the one of the interactive theorem provers, develops proof assistants to develop (and check) proofs using computers. Two big successes are the formal proof of the four colour theorem [7] and of the Kepler conjecture [8]. Using a computer has many advantages: it is easy to test variants and the proofs are checked, meaning that the claimed theorems are really theorems. The main drawback is that writing a proof accepted by the proof assistant is tedious and requires expertise.

The goal of this master’s thesis is to start the formalizing of the rich ATP metatheory. Our basis is Weidenbach’s forthcoming book [9] and our tool is Isabelle (Section 2). This proof assistant has been used for pure mathematics (the Prime Number Theorem proved [10]), for system verification (the seL4 micro-kernel [11]), and programming languages (the formalisation of Java [12, 13]).

During this master’s thesis, I formalized most of Chapter 2 of Weidenbach’s book about inference systems for propositional (or ground) logic (Section 3). Ground formulas are important because the results are “lifted” to first order logic: non ground formulas can be seen as (possibly infinite) sets of grand formulas. To quote Weidenbach: “Everything interesting happens at the ground level” (private communication).

Before applying the calculi, we must normalise the formulas (Section 4). Then we present resolution (Section 5). After that we present DPLL (Section 6) and the implementation of a simple verified SAT solver in OCaml. Finally, we present the formalisation of CDCL (Section 7).

The development is publicly available in a git repository[1] and can be checked using Isabelle [14] version 2015.

2 The Proof Assistant

Isabelle is the prover we use in this formalisation. We describe here the prover (Section 2.1), then how to do definitions (Section 2.2), before describing the Isabelle proofs (Section 2.3).

2.1 General Presentation

Isabelle [14] is a generic framework for interactive theorem proving: the use of a meta-logic *Pure* allows a formalisation for different logics and axiomatisations. The built-in meta-logic Pure is an intuistionistic fragment of higher-order logic. Isabelle is based on the ideas of LCF [15]: every proof goes through a small trusted kernel. It is written in Standard ML, and has an Isabelle/jEdit interface through the asynchronous PIDE interface [16].

HOL, *higher order logic* [17], is the most developed logic in Isabelle: it is based on typed higher-order logic with ML-style rank-1 polymorphism and Haskell-style axiomatic type classes. On the type level, we have either base types (like *nat*), type constructors (for example *bool list* to define lists with elements of type *bool*) or functions (for example $nat \Rightarrow bool$ is a function going from type *nat* to *bool*). $'a$ is a base type that can be instantiated later on, for example with a natural number. On the term level, HOL defines axiomatically *bool*, and the constants $True :: bool$ (i.e. a constant named *True* of type *bool*), $False :: bool$, $= :: 'a \Rightarrow 'a \Rightarrow bool$ and the connectives and $(\wedge)$, or $(\vee)$, not $(\neg)$. HOL is embedded into the meta-logic Pure using $Trueprop :: bool \Rightarrow prop$, that translates from *bool* (defined at the HOL level) to *prop*. *Trueprop* is omitted when printing, allowing to fully ignore the difference between *bool* and *prop*. There is a slight difference between the HOL level with the $\forall$, $\longrightarrow$ and $=$ and their Pure equivalent with $\bigwedge$, $\Longrightarrow$ and $\equiv$, but the differences do not matter in this report. Other logics than HOL include Isabelle/ZF (based on Zermelo-Fraenkel set theory): it is defined using Pure.

In HOL, functions can be curried ($f\ x\ y$ instead of $f(x, y)$), and usual binary operators can be used with the infix notation, like the plus operator $op\ +$ in $2 + 2$, instead of $plus\ 2\ 2$. Function applications bind stronger that infix operators; thus $f\ x + g\ y$ should be read $(f\ x) + (g\ y)$ (the addition of two function calls, $f\ x$ and $g\ y$). Types are annotated with *type classes* to add properties about them: for example in an expression $a + b$, a is inferred of type $'a$, but of type class *plus* to express the property that there is an addition over $'a$. If there is the constrain on a type $'a$ to have type class *plus*, then you can only instantiate $'a$ with a type that has an plus operator. The polymorphism (i.e. working with $'a$) allows to prove things generally (eventually by adding type classes constrains) and instantiate the types later.

The main proof method in HOL is the *simplifier*, which uses equation as oriented rewrite rules on the goals, including conditional rewriting (e.g. $ys = [] \Longrightarrow rev\ ys = z$ will be rewritten into $ys = [] \Longrightarrow rev\ [] = z$; then the simplifier will use the lemma that says $rev\ [] = []$ to get $ys = [] \Longrightarrow [] = z$). The list of lemmas that are used can be extended by the user's lemmas.

One very useful tool in Isabelle/HOL is Sledgehammer [18]: it translates the theorems of Isabelle and exports them to automated provers like CVC4 [5], E [2], SPASS [3], Vampire [4] and Z3 [6]. Then these automated provers try to find a proof: if one is found, then the proof can

[1] https://bitbucket.org/zmaths/formalisation-of-ground-inference-systems-in-a-proof-assistant/

be reconstructed, either in a detailed fashion or using a built-in tactic with the used theorems as arguments. Sledgehammer eases finding proofs, since it avoids looking for easy proofs and remembering the names of hundreds of theorems in the standard library. As of now, none of the formalised theorems have been solved directly by Sledgehammer.

There are a lot of lemmas in either the standard library that is included in Isabelle or the Archive of Formal Proofs [19]. The automation is very developed when working on list and set and is improving when working on multisets. Some related developments in Isabelle include Isabelle Formalisation of Rewriting (IsaFoR) [20]: it is a formalisation of term rewrite systems and of various termination proving tools.

We have used the HOL logic in this master's thesis. We will now give more details about how to define function in Isabelle and write proofs.

2.2 Adding Definitions

Definitions introduces a new theorem to context between the defined symbol and the actual definition. This approach does not introduce axioms and so do not introduce inconsistencies. New definitions are defined using already defined types or terms. There are two levels of definition: types (even types with constructors) and terms.

2.2.1 Types

The primitive way to define a type is to use the **typedef** command: a type is isomorphic to a non-empty set (defined using a type, that is already defined). For example we can define the type of all natural numbers larger than 2:

typedef *my-type* = $\{(n::nat).\ n > 2\}$
by *auto*

where **by** *auto* prove that the type is inhabited.

To define types inductively, the command **datatype** defines inductive and mutually recursive datatypes specified by their constructor

datatype $M :: \tau = C_1\ \overline{x_1} | \cdots | C_\ell\ \overline{x_\ell}$

For example here is a definition of natural numbers:

datatype *nat* =
  Zero | *Suc nat*

There are to constructors *Zero* that takes on argument and *Suc* that takes exactly one argument. The command **datatype** defines the type using **typedef** and prove that the type is not empty (since every type have to be inhabited in HOL).

2.2.2 Terms

A simple definition is of the form **definition** c :: *type* **where** $c\ \overline{x} = t$. This introduces a new axiom of the form $c \equiv \lambda\overline{x}.\ t$ where $\overline{x}$ are the arguments of the function c. Isabelle ensures that the constant c is fresh, the variables in $\overline{x}$ are distinct and that t does not refer to any undefined variable or undefined types. We can for example have:

definition $K :: {'a} \Rightarrow {'b} \Rightarrow {'a}$ **where** $K\ x\ y = x$

Definitions are opaque in the sense that they are not unfolded by default when inside a theorem. Definitions are not recursive (i.e. you cannot call c while defining c). We will use inductive predicates: an inductive predicate is simply a function to *bool* with some assumptions (including calls to itself):

inductive $p :: \tau$ **where**

$$\begin{array}{llllllll} \text{name}_1\text{:} & Q_{11} & \Longrightarrow & \cdots & \Longrightarrow & Q_{1\ell_1} & \Longrightarrow & p\ t_1 \\ \vdots & \vdots & \Longrightarrow & & \Longrightarrow & \vdots & \Longrightarrow & \vdots \\ \text{name}_n\text{:} & Q_{n1} & \Longrightarrow & \cdots & \Longrightarrow & Q_{n\ell_n} & \Longrightarrow & p\ t_n \end{array}$$

where p must be fresh. There are some syntactic restrictions on the rules to ensure monotonicity. Internally the Knaster-Tarski theorem is used with a fixed-point equation to show the existence of a least fixpoint. For example, the following is a definition of the predicate *even*:

inductive *even* :: *nat* ⇒ *bool* **where**
even0: *even 0* |
even-SS: *even n* ⟹ *even* (*Suc* (*Suc n*))

where *Suc* :: *nat* ⇒ *nat* and *0* :: *nat* are the two constructors for natural numbers. The associated fixpoint equation is

$$even\ a = (a = 0 \vee (\exists\, n.\ a = Suc\ (Suc\ n) \wedge even\ n))$$

Contrary to **inductive** definitions where no termination is required and no evaluation is possible in general, when defining a function, termination must be proved. Otherwise a non-terminating definition of the form $f\ x = 1 + f\ x$ where $f{::}nat \Rightarrow nat$ is possible, and then you can deduce $0 = 1$ (by subtracting $f\ x$), which is *False* and allows to prove anything. When the termination is based on a structural decrease, **primrec** can prove automation automatically:

primrec *plus-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
plus-nat 0 n = *0* |
plus-nat (*Suc m*) *n* = *Suc* (*plus-nat m n*)

Here, the number of constructors *Suc* is decreasing for each recursive call. This is a structural decreasing and **primrec** is able to prove termination automatically.

It is also possible to use **fun** that tries a few invariants to show that the argument is decreasing (e.g. the *size* of the formula). **fun** is powerful enough to prove the termination of the Ackerman function, while **primrec** is not:

fun *ack* :: *nat* ⇒ *nat* ⇒ *nat* **where**
ack 0 n = *n* + *1* |
ack (*Suc m*) *0* = *ack m 1* |
ack (*Suc m*) (*Suc n*) = *ack m* (*ack* (*Suc m*) *n*)

If the proof of the termination is more complicated, one can use **function**, which generates three goals: termination, pattern completeness and non-overlapping patterns. An example will be presented in Section 6.4

2.3 Isabelle Proofs

Isabelle proofs are done using some proof methods: they do low level work and go through the (trusted) kernel of Isabelle. Some important tactics are *auto* that uses the simplifier with some knowledge about logic and *blast* that is specialised on logical formulas. *rule th* unifies the conclusion of *th* with the goal: the premises of the goal remains to show.

These proof methods can be combined using two proof styles: forward and backward proofs. The backward proof consists in going from the goal, unifying the conclusion of some theorems with the actual goal. After that, the assumption of the theorems are the new goals. This approach is called **apply**-style. For example, the proof that four is even:

```
lemma even (Suc (Suc (Suc (Suc 0))))
  apply (rule even-SS)
  apply (rule even-SS)
  apply (rule even0)
  done
```

After stating the theorem to prove, the goal is the theorem: *even* (*Suc* (*Suc* (*Suc* (*Suc* *0*)))). We apply the theorem *even* (*Suc* (*Suc* *0*)) $\Longrightarrow$ *even* (*Suc* (*Suc* (*Suc* (*Suc* *0*)))): using theorem *even* *n* $\Longrightarrow$ *even* (*Suc* (*Suc* *n*)), `rule` unifies *n* with *Suc* (*Suc* *0*). It remains to show that *even* (*Suc* (*Suc* *0*)). Then the second **apply** applies theorem *even* *0* $\Longrightarrow$ *even* (*Suc* (*Suc* *0*)). It remains to show that *even 0*. This is true by theorem *even 0*, i.e. the definition of *even*: this theorem has no condition, so our proof is finished.

The other style is forward: you go from the assumptions (if there are some) to the conclusion. This approach is used in Isabelle's *Intelligible semi-automated reasoning* (Isar) [21]. The aim of this language is to be close to the one used by mathematicians.

```
lemma even (Suc (Suc (Suc (Suc 0))))
proof –
  have even 0 by (rule even0)
  then have even (Suc (Suc 0)) by (rule even-SS)
  then show even (Suc (Suc (Suc (Suc 0)))) by (rule even-SS)
qed
```

have introduces a fact, that has to be proved. The proof after can discharged by for example **by** followed by a tactic call. The keyword **then** allows to use previous conclusion, while reasoning on the next one. **show** introduces a fact that is one of the goals needed to prove the theorem.

We have introduced the proof assistant we used during this formalisation; we will now speak about the logic we have formalised in Isabelle.

3 Clausal Logic

In this section we define formally the logic, first defining the syntax (Section 3.1), then the associated semantics (Section 3.2).

3.1 Syntax

We consider a countable and non-empty set Σ. Instead of using a set, we use a type (that we will call $'v$). The non-emptness is required by the type definition in Isabelle. We do not translate the countability constrain into Isabelle, and will show exactly where this constrain is needed. We then define inductively the set $Prop(\Sigma)$ of propositional formulas over a *signature* Σ (usually the alphabet or the words, i.e. countable sets, are used):

datatype $'v$ *propo* =
  FT — the truth symbol |
  FF — the false symbol |
  FVar $'v$ — where $v \in \Sigma$, called atom |
  FNot $'v$ *propo* — is the negation |
  FAnd $'v$ *propo* $'v$ *propo* — is the conjunction symbol |
  FOr $'v$ *propo* $'v$ *propo* — is the disjunction symbol |
  FImp $'v$ *propo* $'v$ *propo* — is the implication symbol |
  FEq $'v$ *propo* $'v$ *propo* — is the equivalence symbol

FT, *FF* and *FVar x* are called *base terms*. *FAnd* is a symbol of the logic we are defining, while $\wedge$ is the conjunction a symbol of the logic in Isabelle. An atom a or its negation *FNot a* is called a *literal*. We will omit the parentheses, assuming the negation symbol binds more strongly than the binary operations.

3.2 Semantics

We are working in classical logic, thus there are two truth values: "true" (*True* in Isabelle or *1* in Weidenbach's presentation) and "false" (*False* in Isabelle or *0*). We define a *valuation*: it is a mapping $\mathcal{A}$: $\Sigma \to \{True, False\}$. A *partial* valuation is a function such that some atoms are not mapped to a value. We extend the valuation over $Prop(\Sigma)$. The standard definition consists in using values *1* and *0*, but to ease the formalisation we have translated the logic into Isabelle's logic:

	Isabelle definition	Weidenbach's definition
$\mathcal{A} \models FT$	*True*	*1*
$\mathcal{A} \models FF$	*False*	*0*
$\mathcal{A} \models FVar\ v$	$\mathcal{A}\ v$	$\mathcal{A}\ v$
$\mathcal{A} \models FNot\ \varphi$	$\neg\ \mathcal{A} \models \varphi$	$1\ -\ \mathcal{A} \models \varphi$
$\mathcal{A} \models FAnd\ \varphi_1\ \varphi_2$	$\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2$	$\max(\mathcal{A} \models \varphi_1, \mathcal{A} \models \varphi_2)$
$\mathcal{A} \models FOr\ \varphi_1\ \varphi_2$	$\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2$	$\min(\mathcal{A} \models \varphi_1, \mathcal{A} \models \varphi_2)$
$\mathcal{A} \models FImp\ \varphi_1\ \varphi_2$	$\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2$	if $\mathcal{A} \models \varphi_1$ then $\mathcal{A} \models \varphi_2$ else 1
$\mathcal{A} \models FEq\ \varphi_1\ \varphi_2$	$(\mathcal{A} \models \varphi_1) \longleftrightarrow (\mathcal{A} \models \varphi_2)$	$(\mathcal{A} \models \varphi_1) = (\mathcal{A} \models \varphi_2)$

We interpret the logic we are formalising (with the *FAnd*) into Isabelle's logic (with the $\wedge$). This simplifies the proofs (for example we do not have to show that the only possible values are *0* and *1*). A formula φ entails ψ, written $\varphi \models f\ \psi$ if for all valuation $\mathcal{A}$ such that $\mathcal{A} \models \varphi$, then $\mathcal{A} \models \psi$. A formula ψ is *satisfiable* if there is a model of ψ. It is called *valid*, written $\models e\ \varphi$, if every mapping is a model of φ: for example *FOr* φ (*FNot* φ) is valid. Another important difference between the paper proofs and the Isabelle proofs is the use of overloaded symbols: $\models$ is used with various types to mean $\models e$, $\models f$ or the valuation $\models$ as defined in the previous table. Isabelle has overloading (meaning that we could have used a single symbol), but it is usually better to be explicit in a formal context. Overloading also makes type inference more difficult, meaning that we have to add type annotations.

A partial valuation can also be seen as a set of literals either positive either negative instead of defining the following partial valuation $\mathcal{A}$: given a set V, for every atom C, $\mathcal{A}(C) = 1$ if $C \in V$, 0 if *FNot* $C \in V$, unspecified otherwise. This set is called the *interpretation*.

Here is a theorem giving a relation between $\models f$ and $\models$. We give a full Isabelle proof and a full paper proof, showing the parallel in the proofs.

Textbook Theorem 1 (Deduction Theorem)**.** $\varphi \models f\ \psi \longleftrightarrow\ \models e\ (\mathit{FImp}\ \varphi\ \psi)$

Proof. ($\Longrightarrow$) Suppose that ϕ entails ψ and let $\mathcal{A}$ be a Σ-valuation. We have to show that $\mathcal{A} \models \mathit{FImp}\ \varphi\ \psi$. If $\mathcal{A}\ \varphi = 1$, then we also have that $\mathcal{A}\ \psi = 1$, thus $\mathcal{A}\ (\mathit{FImp}\ \varphi\ \psi) = \mathit{max}\ (1 - \mathcal{A}\ \varphi)\ (\mathcal{A}\ \psi)$. Otherwise, $\mathcal{A}\ \varphi = 0$, thus $\mathcal{A}\ (\mathit{FImp}\ \varphi\ \psi) = \mathit{max}\ (1 - \mathcal{A}\ \varphi)\ (\mathcal{A}\ \psi){=}1$. We have finally that in both cases $\mathcal{A} \models \mathit{FImp}\ \varphi\ \psi$.

($\Longleftarrow$) Let $\mathcal{A}$ be an arbitrary valuation: $\mathcal{A} \models \mathit{FImp}\ \varphi\ \psi$, i.e. $\mathcal{A}\ (\mathit{FImp}\ \varphi\ \psi) = 1 = \mathit{max}\ (1 - \mathcal{A}(\varphi))\ (\mathcal{A}(\psi))$. If $\mathcal{A}\ \varphi = 0$, then $\mathcal{A}\ \varphi = 0$, otherwise $\mathcal{A}\ \varphi = 1$ and necessary $\mathcal{A}\ \varphi = 1$. So in both cases $\varphi \models f\ \psi$. ☐

Here is a full detailed Isabelle proof. This proof is close to previous proof, except it uses the Isabelle definition instead of arithmetic on *0* and *1*. We do only want to "give a flavour" of Isabelle proofs: the reader is not expected to make fully sense of it.

```
theorem φ ⊨f ψ ⟷ ⊨e (FImp φ ψ)
proof
  assume H: φ ⊨f ψ
  show ⊨e (FImp φ ψ)
  unfolding entails-def
  proof
    fix 𝒜
    { assume 𝒜 ⊨ φ
      then have 𝒜 ⊨ ψ using H unfolding evalf-def by metis
      then have 𝒜 ⊨ FImp φ ψ by auto
    }
    also {
      assume ¬ 𝒜 ⊨ φ
      hence 𝒜 ⊨ FImp φ ψ by auto
    }
    ultimately show 𝒜⊨ FImp φ ψ by blast
  qed
next
  assume H: ⊨e (FImp φ ψ)
  show φ ⊨f ψ
    proof (rule ccontr)
      assume ¬φ⊨f ψ
```

    then obtain A **where** $A \models \varphi \wedge \neg A \models \psi$ **using** *evalf-def* **by** *metis*
    hence $\neg\ A \models$ *FImp* $\varphi\ \psi$ **by** *auto*
    then show *False* **using** *H entails-def* **by** *blast*
  qed
qed

There are two blocks between the separated by **next**: the first block is the implication and the other the converse (as in the paper proof). The keywords of the Isabelle proof are close to the words used in the other version.

We have fully detailed the proof to show that the Isabelle proof can be very close to the paper proof. A shorter is possible using the simplifier and the definition of $\models e$ and $\models f$:

theorem $\varphi \models f\ \psi \longleftrightarrow\ \models e$ (*FImp* $\varphi\ \psi$)
 by (*simp add*: *evalf-def entails-def*)

This is an atypical example: the detailed paper proof is longer than the Isabelle version. Usually, it is the other way around.

4 Normal Forms

Before trying to solve the actual problem, we must normalise the set of formulas N that we are considering. We will formally define two normal forms (Section 4.1), then introduce transitions systems (Section 4.2) and show how to define them in Isabelle (Section 4.3).

4.1 Definition of Two Normal Forms

We give here the definitions of two normal form as in Weidenbach's book. The idea of this two normal forms is to remove the *FImp* and *FEq*.

Definition 1 (CNF, clauses)**.** *A formula is in* conjunctive normal form *(CNF) if it is a conjunction of clauses (i.e. of disjunction of literals):* $\bigwedge_i \bigvee_j L_{i,j}$ *where* $L_{i,j}$ *are literals and* $\bigvee_j L_{i,j}$ *is a clause.*

Definition 2 (DNF)**.** *A formula is in* disjunctive normal form *(DNF) if it is a disjunction of conjunction of literals:* $\bigvee_i \bigwedge_j L_{i,j}$ *where* $L_{i,j}$ *are literals.*

An interesting property of this representation is that $C \subset C'$ means that C' is more general: the satisfiability of C implies the one of C'; each valuation satisfying C satisfies also C and C' (written $\{C, C'\}$ seen as a set of clauses). We will write this as $I \models C$ implies $I \models s\ \{C,\ C'\}$.

General clausal propositional are related to the normal form by the following theorem:

Verified Theorem 3. *Each propositional formula can be transformed into an equivalent CNF form.*

Proof. First we recursively transform $\phi \to \psi$ and $\phi \leftrightarrow \psi$ into $\neg\phi \vee \psi$ and $(\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi)$. Then we reorganise the $\wedge$ and the $\vee$ (see Algorithm 4.1). This algorithm preserves not only validity (the same models satisfy the formula before and after normalisation), but also it is also an equivalence transformation. It is an algorithm with exponential complexity and faster algorithms exist in practice. $\square$

Algorithm 4.1: Transformation into CNF form

```
Data: A formula $\varphi$ without $\rightarrow$ nor $\leftrightarrow$
Result: A formula $\varphi'$ equivalent to $\varphi$ in CNF
Algorithm cnf($\varphi$)
    switch $\varphi$ do
        case $\top, \bot, \neg\top, \neg\bot$
            return $\varphi$
        end
        case $\phi \wedge \psi$
            $\phi_1 \wedge \cdots \wedge \phi_n :=$ cnf ($\phi$)
            $\psi_1 \wedge \cdots \wedge \psi_m :=$ cnf ($\psi$)
            return $\phi_1 \wedge \cdots \wedge \phi_n \wedge \psi_1 \wedge \cdots \wedge \psi_m$
        end
        case $\phi \vee \psi$
            $\phi_1 \wedge \cdots \wedge \phi_n :=$ cnf ($\phi$)
            $\psi_1 \wedge \cdots \wedge \psi_m :=$ cnf ($\psi$)
            return $(\phi_1 \vee \psi_1) \wedge \cdots \wedge (\phi_1 \vee \psi_m) \wedge \cdots \wedge (\phi_n \vee \psi_1) \wedge \cdots \wedge (\phi_n \vee \psi_m)$
        end
        case $\neg(\phi \vee \psi)$
            return cnf($\neg\phi \wedge \neg\psi$)
        end
        case $\neg(\phi \wedge \psi)$
            return cnf($\neg\phi \vee \neg\psi$)
        end
    endsw
```

After that, we can do some simplification on the translation into the CNF like transforming $P \vee \neg P$ into $\top$ and $\bot \vee Q$ into $\bot$. These simplifications are not necessary but improve efficiency.

The presentation as a transition system allows to prove various refinement and implementations that changes the order in which the transformation are done. Changing the order of the transformation lead to better performance for example. The Algorithm 4.1 is one of the possible orders: the innermost formulas are transformed first, and then the whole formula is transformed. For example in *FAnd* φ ψ, φ and ψ are transformed before *FAnd* φ ψ is transformed.

4.2 Positions, Transition Systems

To refer to a subformula, we define the *position* as list over $\{L, R\}$ where R means right and L means left (L is also the default value if there is no right formula, e.g. ψ is on the left in *FNot* ψ). The empty list [] means that we are considering the actual level. The set of all positions in a formula is given by (where $L \cdot p$ means that L is added at the beginning of the list p):

$$\begin{array}{lcl}
pos\ FF & = & \{[]\} \\
pos\ FT & = & \{[]\} \\
pos\ (FVar\ x) & = & \{[]\} \\
pos\ (FAnd\ \varphi\ \psi) & = & \{[]\} \cup \{L \cdot p \mid p \in pos\ \varphi\} \cup \{R \cdot p \mid p \in pos\ \psi\} \\
pos\ (FOr\ \varphi\ \psi) & = & \{[]\} \cup \{L \cdot p \mid p \in pos\ \varphi\} \cup \{R \cdot p \mid p \in pos\ \psi\} \\
pos\ (FEq\ \varphi\ \psi) & = & \{[]\} \cup \{L \cdot p \mid p \in pos\ \varphi\} \cup \{R \cdot p \mid p \in pos\ \psi\} \\
pos\ (FImp\ \varphi\ \psi) & = & \{[]\} \cup \{L \cdot p \mid p \in pos\ \varphi\} \cup \{R \cdot p \mid p \in pos\ \psi\} \\
pos\ (FNot\ \varphi) & = & \{[]\} \cup \{L \cdot p \mid p \in pos\ \varphi\}
\end{array}$$

For example φ is at position $[L, R, L]$ in $FAnd\ (FEq\ FT\ (FNot\ \varphi))\ FF$. Using the position, removing the equivalence symbols can become: transform $FEq\ \varphi_1\ \varphi_2$ into $FAnd\ (FImp\ \varphi_1\ \varphi_2)$ $(FImp\ \varphi_2\ \varphi_1)$ at any possible position with an *FEq*. This is not deterministic anymore (we are not giving an order on the possible positions) and we do not care about the exact chosen path, but only on properties on the result (that do not have to be unique). The heuristics used in practise try to find the shortest path, but we show only properties on the final state (Figure 1), whatever the path is.

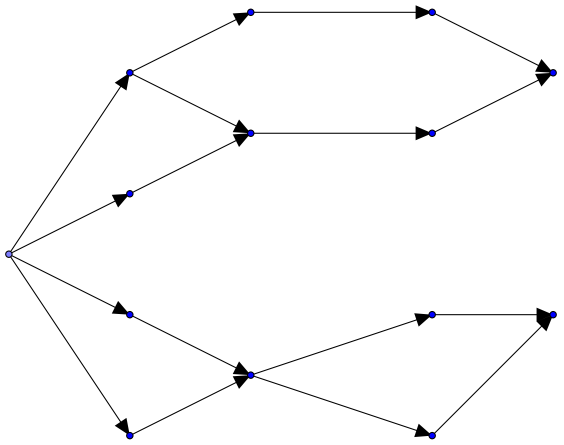


Figure 1: Transition system: each arrow represent a possible transition. The points represents states: there are two final states (the two states on the right, without leaving arrow). While an algorithm will chose deterministically one of the different paths, our presentation as a transition system means that we show properties on all possible paths and all the possible final states.

The transformation defined as a transition system is defined as $\chi[FEq\ \varphi_1\ \varphi_2]_p \Longrightarrow_{elim\text{-}eq} \chi[FAnd\ (FImp\ \varphi_1\ \varphi_2)\ (FImp\ \varphi_2\ \varphi_1)]_p$ where $\chi[\varphi]_p$ means that φ is at position p in χ and nothing else has changed during the transformation $\Longrightarrow_{elim\text{-}eq}$. This vision as a transition system is more flexible than an algorithm: whatever order on the rewriting is more convenient, it does not matter for the proof of the transition system.

The definition of a transition system consists in giving all possible transitions: in Figure 1, it corresponds to all the possible arrows. The underlying non-deterministic algorithm is simply: do a transition among all the possible ones, as long as possible. In our the example, whatever the order of applying replacing the *FEq* symbols, we will get a formula without *FEq* symbols. The algorithm terminates when we are in a state where no transition is possible: this is called a *final state*. No unicity is required on the final states.

4.3 Formalisation in Isabelle

We used a presentation as a transition system and not as a term rewrite system: in the book, rewrite systems have been introduced in a previous chapter of the book. We could have formalised rewrite system using IsaFoR [20], but it is an overkill for what we have formalised. We have introduced a transition system that do rewriting, then we lift this rewriting to rewriting one of the subformulas.

Algorithm 4.2: Basic CNF/DNF transformation (see rules in Figure 2)

```
  Data: A propositional formula $\varphi$
  Result: A propositional formula $\varphi'$ equivalent to $\varphi$ in CNF or DNF form)
1 apply rule elim-equiv as long as possible
2 apply rule elim-imp as long as possible
3 apply rule elimTB as long as possible
4 apply rule pushNeg as long as possible
5 apply rule pushConj as long as possible for CNF
6 apply rule pushDisj as long as possible for DNF
```

4.3.1 CNF, DNF Definition

Contrary to the Weidenbach's definitions of CNF and DNF, our operators are only binary connectives, so we cannot express the definitions exactly as in Definitions 1 and 2. More precisely, an implicit generalisation from binary to n-ary is used in Weidenbach's book, because there are semantically the same. Isabelle does not allow these changes (since the formulas are synctatically different), so we have only binary operators. For example, *FAnd* (*FAnd* a b) (*FAnd* c d) would be written $a \wedge b \wedge c \wedge d$, relying on the associativity of *FAnd*.

If we look at the definitions, there are there constrains: firstly in the innermost their are only literals; secondly this literals can be connected by the connective *FOr* (respectively *FAnd*), finally these groups are connected by *FAnd* (respectively by *FOr*). In Isabelle we will separate these three constrains: we will relax the constrain on literals and allow base terms (*FT*, *FT* or *FVar* v for an atom v). These can be grouped using a connective c that will be instantiate depending on whether we define CNF or DNF:

- we can have the simple terms directly:

$$\frac{\textit{simple}\ \varphi}{\textit{grouped-by}\ c\ \varphi}$$

- or we can have the negation of a simple term:

$$\frac{\textit{simple}\ \varphi}{\textit{grouped-by}\ c\ (\textit{FNot}\ \varphi)}$$

- or we can group two groups using the c connective. The *wf-conn* c l verifies that it is a real term.

elim-equiv: $\textit{elim-equiv}\ (\textit{FEq}\ \varphi\ \psi)\ (\textit{FAnd}\ (\textit{FImp}\ \varphi\ \psi)\ (\textit{FImp}\ \psi\ \varphi))$

elim-imp: $\textit{elim-imp}\ (\textit{FImp}\ \varphi\ \psi)\ (\textit{FOr}\ (\textit{FNot}\ \varphi)\ \psi)$

elimTB: remove all the unused FT and FF:

$\textit{elimTB}\ (\textit{FAnd}\ \varphi\ \textit{FT})\ \varphi$

$\textit{elimTB}\ (\textit{FAnd}\ \textit{FT}\ \varphi)\ \varphi$

$\textit{elimTB}\ (\textit{FAnd}\ \varphi\ \textit{FF})\ \textit{FF}$

$\textit{elimTB}\ (\textit{FAnd}\ \textit{FF}\ \varphi)\ \textit{FF}$

$\textit{elimTB}\ (\textit{FOr}\ \varphi\ \textit{FT})\ \textit{FT}$

$\textit{elimTB}\ (\textit{FOr}\ \textit{FT}\ \varphi)\ \textit{FT}$

$\textit{elimTB}\ (\textit{FOr}\ \varphi\ \textit{FF})\ \varphi$

$\textit{elimTB}\ (\textit{FOr}\ \textit{FF}\ \varphi)\ \varphi$

$\textit{elimTB}\ (\textit{FNot}\ \textit{FT})\ \textit{FF}$

$\textit{elimTB}\ (\textit{FNot}\ \textit{FF})\ \textit{FT}$

pushNeg: push the *FNot* at the innermost:

$\textit{pushNeg}\ (\textit{FNot}\ (\textit{FAnd}\ \varphi\ \psi))\ (\textit{FOr}\ (\textit{FNot}\ \varphi)\ (\textit{FNot}\ \psi))$

$\textit{pushNeg}\ (\textit{FNot}\ (\textit{FOr}\ \varphi\ \psi))\ (\textit{FAnd}\ (\textit{FNot}\ \varphi)\ (\textit{FNot}\ \psi))$

$\textit{pushNeg}\ (\textit{FNot}\ (\textit{FNot}\ \varphi))\ \varphi$

pushConj: $\textit{pushConj} = \textit{push-conn-inside}\ \textit{CAnd}\ \textit{COr}$

pushDisj: $\textit{pushDisj} = \textit{push-conn-inside}\ \textit{COr}\ \textit{CAnd}$

Figure 2: Transformation rules

$$\frac{\textit{grouped-by}\ c\ \varphi \qquad \textit{grouped-by}\ c\ \psi \qquad \textit{wf-conn}\ c\ [\varphi,\ \psi]}{\textit{grouped-by}\ c\ (\textit{conn}\ c\ [\varphi,\ \psi])}$$

After that we make groups withe the other connective that will be instantiated depending on whether we will define CNF or DNF. An inner group is also an outer group and we can combine outer group using the correct connective:

- either we have a group:

$$\frac{\textit{grouped-by}\ c\ \varphi}{\textit{super-grouped-by}\ c\ c'\ \varphi}$$

- or we make groups:

$$\frac{\textit{super-grouped-by}\ c\ c'\ \varphi \qquad \textit{super-grouped-by}\ c\ c'\ \psi \qquad \textit{wf-conn}\ c\ [\varphi,\ \psi]}{\textit{super-grouped-by}\ c\ c'\ (\textit{conn}\ c'\ [\varphi,\ \psi])}$$

It is important to notice is that *FF* is a correct CNF and DNF formula, although there is a no single literal in *FF*. This case is not explicitly stated in the definition of CNF and DNF, because an empty formula is not very interesting, but the question arises when doing the Isabelle proof. Thus we use a predicate *no-T-F-except-top-level* that checks that there is no *FT* nor *FF*, except on top-level. We can now combine all these properties:

- *is-cnf* $\varphi \equiv$ *is-conj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ where *is-conj-with-TF* is an abbreviation for *super-grouped-by COr CAnd*

- *is-dnf* $\varphi \equiv$ *is-disj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ where *is-disj-with-TF* is an abbreviation for *super-grouped-by CAnd COr*

4.3.2 Transition System

We have formalised the approach as a transition system without using explicitly the path that leads to the rewritten formula: given a rewrite relation r, rewriting means that at each level either the actual formula is rewritten or a single one of the arguments has changed.

To make a shorter definition, instead of writing every formula case (*FAnd*, *FOr*,...), we made a higher representation: a formula is a connective (*CAnd*, *COr*,...) and a list of arguments. This allows to have an shorter representation, but we have to ensures that the list really corresponds to a term, thus the predicate *wf-conn* c φs that ensures the number of arguments of c is compatible with the number of elements in φs, e.g. the connective *CAnd* corresponding to *FAnd* has two arguments exactly.

The predicate *propo-rew-step* r φ ψ means that φ is rewritten in ψ by the relation r. It is defined inductively as follows:

- We use a rule representation: on the above you have the assumption to fulfil (here simply r φ ψ) to show the term below the line.

$$\frac{r\ \varphi\ \psi}{\textit{propo-rew-step}\ r\ \varphi\ \psi}$$

- Otherwise, rewriting one of the argument is enough:

$$\frac{\textit{propo-rew-step}\ r\ \varphi\ \varphi' \qquad \textit{wf-conn}\ c\ (\psi s\ @\ (\varphi \cdot \psi s'))}{\textit{propo-rew-step}\ r\ (\textit{conn}\ c\ (\psi s\ @\ (\varphi \cdot \psi s')))\ (\textit{conn}\ c\ (\psi s\ @\ (\varphi' \cdot \psi s')))}$$

  where @ is the append function. It is a complicated formula (since it is general to enough to mach every possible rewriting), but it means simply that a single one of the arguments has been rewritten. For example if you have *propo-rew-step* r (*FAnd* φ ψ) (*FAnd* φ' ψ') then either r (*conn CAnd* $[\varphi, \psi]$) (*conn CAnd* $[\varphi', \psi']$), or one of the arguments has changed: φ into ψ and $\varphi' = \psi'$; or φ' into ψ' and $\varphi = \psi$.

This abstraction over rewriting is independent of the considered transformation. The link between the path approach and *propo-rew-step* is given by:

Verified Theorem 4. *If propo-rew-step $r\ \varphi\ \varphi'$ then $\exists\,\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ path-to $p\ \varphi\ \psi \wedge$ replace-at $p\ \varphi\ \psi' = \varphi'$.*

The theorem means that whenever we have some transition from φ to φ', then there is some ψ at a path p in φ that is rewritten.

To prove the full transformation of Algorithm 4.2, we iterate the transition relation *propo-rew-step* r to rewrite as long as possible (i.e. until no more step is possible): (*propo-rew-step* r)** is the reflexive transitive closure of a curried predicate.

Definition 5. $r^{\downarrow} = (\lambda\varphi\ \psi.\ (\textit{propo-rew-step}\ r)^{**}\ \varphi\ \psi \wedge (\forall\,\psi'.\ \neg\ \textit{propo-rew-step}\ r\ \psi\ \psi'))$

Each transformation is one call to $^{\downarrow}$. For each of these transformations, we have to show some invariants and we can then compose our transformations to get the full transformation of the Algorithm 4.2 by using the relational composition *op* $\odot$:

$$\textit{cnf-rew} = \textit{elim-equiv}^{\downarrow} \odot \textit{elim-imp}^{\downarrow} \odot \textit{elimTB}^{\downarrow} \odot \textit{pushNeg}^{\downarrow} \odot \textit{pushDisj}^{\downarrow}$$

The operator *op* $\odot$ is defined by $(R \odot S) = (\lambda x\ z.\ \exists\,y.\ R\ x\ y \wedge S\ y\ z)$. In the definition of *cnf-rew*, *cnf-rew* φ ψ means that there are some formulas ϱs such that (*elim-equiv*$^{\downarrow}$) φ ϱ, (*elim-imp*$^{\downarrow}$) ϱ ϱ', (*elimTB*$^{\downarrow}$) ϱ' ϱ'', (*pushNeg*$^{\downarrow}$) ϱ'' ϱ''' and (*pushDisj*$^{\downarrow}$) ϱ''' ψ

We can then prove the following two theorems:

Verified Theorem 6 (Equivalence preservation)**.** *cnf-rew* $\varphi\ \psi \longrightarrow (\forall\,\mathcal{A}.\ (\mathcal{A} \models \varphi) = (\mathcal{A} \models \psi))$

Proof. The idea of the proof is to show that the equivalence is preserved by each transformation used in *cnf-rew*. For *elim-equiv* for example, we have to show that *elim-equiv* $\varphi\ \psi \Longrightarrow \forall\,\mathcal{A}.\ (\mathcal{A} \models \varphi) = (\mathcal{A} \models \psi)$, hence (*elim-equiv*$^{\downarrow}$) $\varphi\ \psi \Longrightarrow \forall\,\mathcal{A}.\ (\mathcal{A} \models \varphi) = (\mathcal{A} \models \psi)$. ☐

Verified Theorem 7 (Correctness)**.** *cnf-rew* $\varphi\ \varphi' \Longrightarrow$ *is-cnf* φ'*.*

4.3.3 Changing the Order of the Rules

It is better to change the order of the rules and to remove the *FT* and *FF* symbols before any other transformation. This leads to shorter formulas, which means that less transitions are needed. The algorithm changes the rules and more elimination are required for *FT* and *FF*. The definition of this removing *elimTBFull* is a bit more complicated: there are more cases, for example replacing *FEq FF* φ into *FNot* φ. The associated reordering does not change the ideas behind the proof and except the reordering (namely removing *FF* and *FT* at the beginning) does not change the other transformations. The new transformation can be defined as:

Definition 8. $\textit{cnf-rew}' \equiv \textit{elimTBFull}^{\downarrow} \odot \textit{elim-equiv}^{\downarrow} \odot \textit{elim-imp}^{\downarrow} \odot \textit{pushNeg}^{\downarrow} \odot \textit{pushDisj}^{\downarrow}$

elimTBFull removes all the true and false symbols, except the one at the top-level (for example for the formula *FImp FT FT*), which cannot be removed.

4.3.4 Alternative CNF Representation

All the proof methods we will present in the next sections assumes that the formula are in CNF. To ease the work, instead of assuming each time that the formulas are in CNF, we will use a different representation. A literal is a positive atom or a negative one:

datatype *'a literal* =
  Pos 'a
| *Neg 'a*

After that a clause is simply a multiset of literals and clauses are set of clauses (of type *'a clause*):

type-synonym *'a clause* = *'a literal multiset*
type-synonym *'v clauses* = *'v clause set*

We use a *type-synonym* to be able to write *'a clause* instead of the multiset version, but we do not use an opaque type. Multisets are written for example $\{\!\{Pos\ P,\ Pos\ P\}\!\}$. All other operators are written the same way as the usual set counterpart ($\in$ for inclusion of an element and $\subset$ for *strict* inclusion).

In this representation, the equivalent *FF* of the empty multiset: we will write it $\bot$. There is no equivalent of *FT*. This is not a real problem: given a set of formulas, we can remove all the true formulas, except if we have only true formulas in our set: in that case, there is no contradiction to find.

We will either write the multiset of literals $\{\!\{L_1,\ \ldots,\ L_n\}\!\}$ (this is the Isabelle representation) or $L_1 \vee \cdots \vee L_n$ (this is the more common notation) for the clauses. Multisets are not ordered, contrary to clauses: $\{\!\{L,\ L'\}\!\} = \{\!\{L',\ L\}\!\}$ while on the syntax level, $(L \vee L') \neq (L' \vee L)$, but thanks to associativity and commutativity of $\vee$, the order in clauses does not matter.

The length of the development is 500 lines of code to define the logic as presented in the previous section. Then there are 300 lines of code that defines abstract transformation and 1 000 lines of code to define the transformations, CNF and DNF. Now we have defined how to normalise the representation of a set of formulas; we will describe the algorithms that we have formalised.

5 The Resolution Calculus

The resolution calculus was invented by Robinson [1]: it does transformation on the sets of clauses to find constrains on the values of the literals. This deductions called inferences will be presented in Section 5.1; then we will describe in more details the inference system we are using in Section 5.2 and finally give an overview of the Isabelle proof (Section 5.3.1).

5.1 Inferences

We try to find a contradiction in N, a set of clauses in CNF. This proof of $\bot$ that we are looking for is a trace of application of inference rules. An inference rule deduces a conclusion $\mathcal{C}$ from a some premises $\mathcal{C}_1, \dots, \mathcal{C}_n$, written:

$$\frac{\mathcal{C}_1 \qquad \cdots \qquad \mathcal{C}_n}{\mathcal{C}}$$

We can also see inferences as transitions: $(N \cup \{C_1, \dots, C_n\}) \Rightarrow (N \cup \{C_1, \dots, C_n\} \cup \{C\})$ is the transition associated to the inference rule. No clause is removed by an inference rule. An inference is said to be *sound*, when $\mathcal{C}$ is entailed by its premises: $\mathcal{C}_1, \dots, \mathcal{C}_n \vDash \mathcal{C}$.

A simple procedure to find a proof of false consists in applying all rules over and over as long as we find new formulas until we find $\bot$. This is a final state for the procedure (we reached false), but it is not necessary a termination state for the rewrite system. The procedure stops when $\bot$ is found, or when the rules do not provide any new information: at this point the set of all the formulas we have found is called *saturated*. An inference system is *refutational complete* if we can always find $\bot$ whenever the given set of formulas is inconsistent.

Remark that the given procedure does not always terminate: if new inferences can always be deduced, then it does not stop. If the procedure is refutational complete, then the procedure is a *semi-decision procedure*: it stops if the set is inconsistent (and a proof is found); otherwise, it can stop (in a state where we know that the set of formula is statisfiable) or not.

5.2 The Rules

We use proof trees in this part instead of a transition system to make the inferences easier to understand and avoid writing the set of known clauses at each step: in a proof tree, this set is the clauses we have started with and the clauses we have deduced by applying the rules.

Let us consider the following rules called *binary resolution with factoring*:

$$\frac{C \vee P \qquad \neg P \vee C'}{C \vee C'}\ \mathsf{Res}$$

(a) Resolution on the literal P

$$\frac{C \vee L \vee L}{C \vee L}\ \mathsf{Fact}$$

(b) Factoring of the literal L

Figure 3: The rules of resolution calculus.

An important point is that in the two rules, C and C' can be the empty clause: for example, the Factorisation rule allows to deduce L from $L \vee L$. The reason is that the $\vee$ based representation is only a representation of the multiset presentation.

The Res rule is a sound inference. It can be easily shown by case distinction on the value of A: if A is true in a model, then $\neg A$ is false and thus D must be true, because the premises are valid. Otherwise, A is false: C must be true.

We consider the set of formulas in CNF: $N = \{\neg A \lor B, \neg B \lor C, A \lor \neg C, A \lor B \lor C \lor C, \neg A \lor \neg B \lor \neg C\}$, from which we want to deduce $\bot$. On one hand we can deduce C:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \lor B \lor C \lor C}{A \lor B \lor C}\,\mathsf{Fact} \qquad A \lor \neg C}{A \lor B \lor A}\,\mathsf{Res}}{B \lor A}\,\mathsf{Factorisation} \qquad \neg A \lor B}{B \lor B}\,\mathsf{Res}}{B} \qquad \neg B \lor C}{C}\,\mathsf{Res}$$

On the other hand, but we can also deduce $\neg C$, by using C and B we have previously proven:

$$\dfrac{\dfrac{\dfrac{C \qquad A \lor \neg C}{A}\,\mathsf{Res} \qquad \neg A \lor \neg B \lor \neg C}{\neg B \lor \neg C}\,\mathsf{Res} \qquad B}{\neg C}\,\mathsf{Res}$$

Thus: $\dfrac{\neg C \qquad C}{\bot}\,\mathsf{Res}$. We apply the rule in a given order and change the rule, on which we apply the rules: if the Factorisation rule is applied over-and-over on the *same* clause $A \lor B \lor C \lor C$, then each time we get the same $A \lor B \lor C$. To prove termination and correctness, we would have to ensure that each rule is not always applied on the same clause.

5.3 Formalisation in Isabelle

We will first describe the formalisation (Section 5.3.1). In this version, clauses are only added to our set of clause, while it can useful to remove some to reduce the search space (Section 5.3.2).

5.3.1 The Calculus

$$\dfrac{\{\!\!\{\mathit{Pos}\ p\}\!\!\} + C \in N \qquad \{\!\!\{\mathit{Neg}\ p\}\!\!\} + D \in N \qquad (\{\!\!\{\mathit{Pos}\ p\}\!\!\} + C,\ \{\!\!\{\mathit{Neg}\ p\}\!\!\} + D) \notin \mathit{already\text{-}used}}{(N,\ \mathit{already\text{-}used}) \Rightarrow_{\mathrm{Res}} (C + D,\ \mathit{already\text{-}used} \cup \{(\{\!\!\{\mathit{Pos}\ p\}\!\!\} + C,\ \{\!\!\{\mathit{Neg}\ p\}\!\!\} + D)\})}\,\mathsf{Res}$$

(a) Resolution on A

$$\dfrac{\{\!\!\{L,\ L\}\!\!\} + C \in N}{(N,\ \mathit{already\text{-}used}) \Rightarrow_{\mathrm{Res}} (C + \{\!\!\{L\}\!\!\},\ \mathit{already\text{-}used})}\,\mathsf{Fact}$$

(b) Factoring on L

Figure 4: The rules of resolution calculus

To prove termination, we need to ensures that no rule is applied twice to the same clauses and contrary to a paper proof we can not add the condition after the definition. More precisely removing a duplicate literal from a clause (rule Fact) terminates, but repeated applications of Res does not necessary, thus we maintain a set called *already-used* containing the pair of premises that we have already used. This slight addition is only added to the proof of the termination theorem and not to the other proof like *soundness* and *completeness*, as if it does not make any difference, while it does.

The rules of the Isabelle version are in Figure 4: with this rules, the case C empty is in the rules, since it is simply C being the empty multiset.

The soundness and completeness theorem we wrong in Weidenbach's book [9], but the idea of the proof is correct:

Textbook Theorem 9 (Wrong Version)**.** *The resolution calculus is sound and complete:*
N is unsatisfiable iff $N \Rightarrow_{\mathrm{Res}}{}^{\star} \{\bot\}$.

Textbook Theorem 9 (Corrected Version)**.** *The resolution calculus is sound and complete:*
N is unsatisfiable iff there is some N such that $N \Rightarrow_{\mathrm{Res}}{}^{\star} N'$ where $\bot \in N'$.

The difference between the wrong and the corrected version of the theorem is that in one case $\bot \in N'$ and in the other case $N' = \{\bot\}$ (the set containing only the empty clause). The theorem is correct if we add simplification rules as we do in the next section, but under the current rules it is not. A corrected version verified in Isabelle is the following theorem:

Verified Theorem 9 (Soundness and Completeness)**.** *If finite* $(fst\ \psi)$ *and* $snd\ \psi = \emptyset$*, then the following equivalence holds:* $\exists \psi'.\ \psi \Rightarrow_{\mathrm{Res}}{}^{**} \psi' \land \bot \in fst\ \psi'$ *if and only if unsatisfiable* $(fst\ \psi)$.

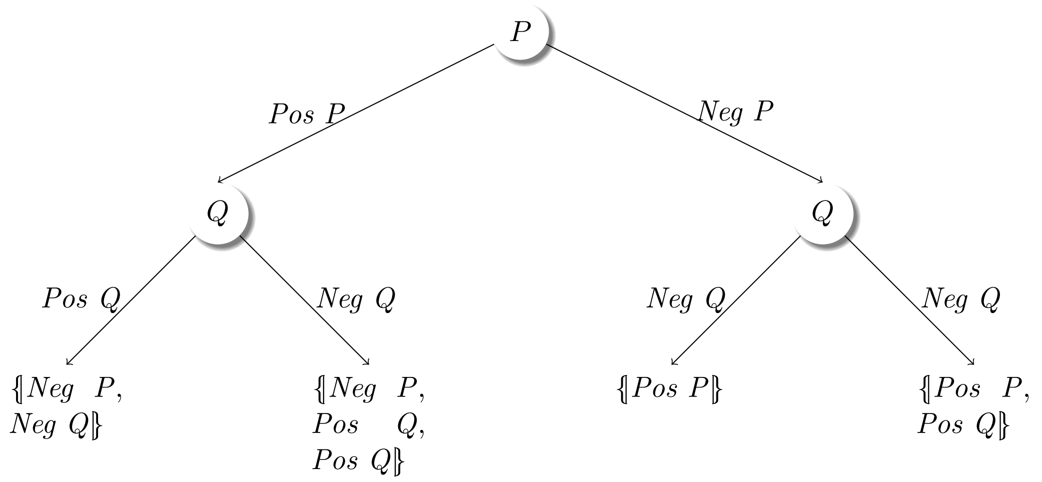
There are two differences between both theorems: first (to prove termination later), we use a set of used clauses, which is initially empty, thus the condition $snd\ \psi = \emptyset$. Moreover we have to assume that the number of clauses is finite since sets are infinite in Isabelle, thus the *finite* $(fst\ \psi)$.

Proof. ($\Longleftarrow$) The converse is a proof by contradiction: assume that N is satisfiable and we have $\psi \Rightarrow_{\mathrm{Res}}{}^{**} \psi'$ such that $\bot \in fst\ \psi'$ is impossible. Each transition $(N, used) \Rightarrow_{\mathrm{Res}} (N', used)$ is such that $N \models ps\ N'$. As $\bot \in \psi'$, we have $N \models p\ \bot$. This is false, since N is satisfiable.

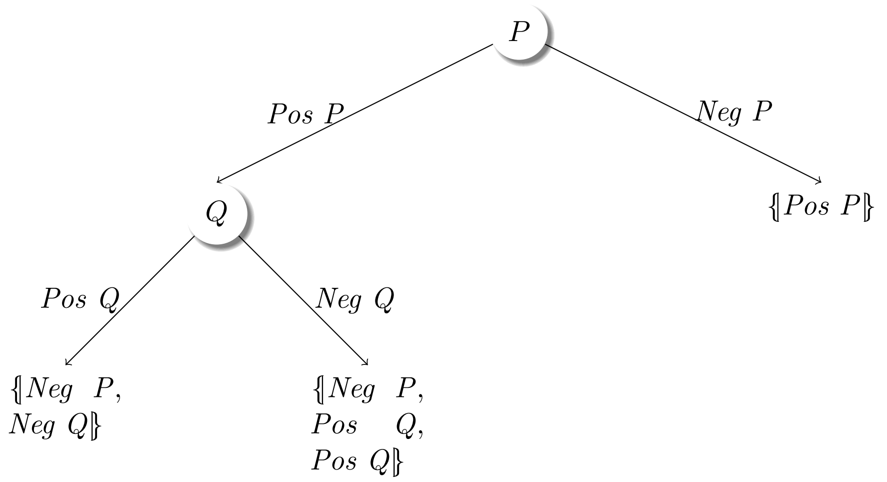
($\Longrightarrow$) We assume that N is unsatisfiable. The idea of the proof of the implication in the theorem is to build a semantic tree and to decrease the size by merging sibling leafs. Each of this merging consists in applying some rules of the calculus.

A semantic tree is a binary tree, such that each node is labelled with an atom and the leafs are labelled with a formula. Given a node marked with the atom l, the literal *Pos* l is true in the left subtree and the literal *Neg* l is true in the right subtree. At the level of the leafs, all these literals are a (possibly partial) valuation. The formula φ at a given leaf is such that the interpretation is I is not a model: $I \not\models \varphi$. For example in Figure 5a, the leftmost leaf is labelled with the formula $\{\!\{Neg\ P,\ Neg\ Q\}\!\}$. The interpretation is $\{\!\{Pos\ P,\ Pos\ Q\}\!\}$, and we have $\{\!\{Neg\ P,\ Neg\ Q\}\!\} \not\models \{\!\{Neg\ P,\ Neg\ Q\}\!\}$.

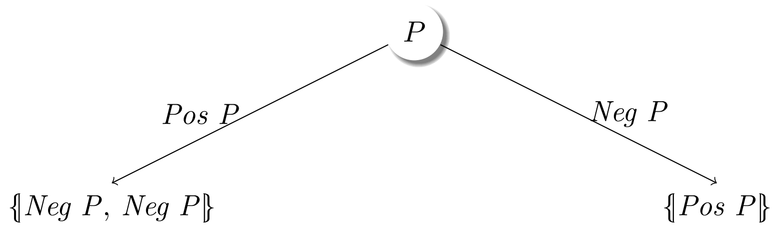
We will now describe the merging of the sibling node on an example, with the following set of clauses: $N = \{\{\!\{Pos\ P\}\!\}, \{\!\{Pos\ P,\ Neg\ P\}\!\}, \{\!\{Neg\ P,\ Pos\ Q,\ Pos\ Q\}\!\}, \{\!\{Neg\ P,\ Pos\ Q,\ Pos\ Q\}\!\}, \{\!\{Neg\ P,\ Neg\ Q\}\!\}, \{\!\{Pos\ P,\ Pos\ Q\}\!\}\}$. The first step is to build a semantic tree associated to the set of clauses. To do so we build a semantic tree such that every path from the node to a leaf is a *total* valuation. As the valuation are total, there is always a clause in N such that the interpretation is not a model. For our set of formulas, a possible semantic tree is in Figure 5a.



(a) Initial semantic tree



(b) Initial semantic tree after merging one sibling node



(c) Initial semantic tree after merging two sibling nodes

$\perp$

(d) Initial semantic tree after merging all sibling nodes

Figure 5: Reduction of the size of a semantic tree, on the set of clauses $N = \{\{\!|Pos\ P|\!\}, \{\!|Pos\ P, Neg\ P|\!\}, \{\!|Neg\ P, Pos\ Q, Pos\ Q|\!\}, \{\!|Neg\ P, Pos\ Q, Pos\ Q|\!\}, \{\!|Neg\ P, Neg\ Q|\!\}, \{\!|Pos\ P, Pos\ Q|\!\}\}$

The Isabelle version consists in taking an element of all the atoms and building the tree recursively:

```
build-sem-tree atms $\psi$ =
  (if atms = {} $\vee$ $\neg$ finite then Leaf
  else
    let m = Min atms in
    let t = build-sem-tree (atms − {m}) $\psi$ in
    Node m t t)
```

The condition *atms* $= \emptyset \vee \neg$ *finite atms* is necessary to prove termination: if we could define without and apply it on an infinite set of atoms, the tree would be infinite, which is impossible, since we are using a datatype. We have to find a way to take an element of the set: we take the minimum of the set. That is why we have added a type class *linorder* on the type of the atoms: it means that there is a linear order of the set. This allows us to take the variables in a given order to build the tree (here taking the minimum of the variables not yet in the tree but in the set for formulas). This is not an issue: as we have only a finite number of literals, we can give each of this literals a number and then use the order given by this number. Moreover in practice we use a words over the alphabet and there is an natural order on it (the lexicographic order).

Now we have built the semantic tree, we can start merging the siblings. First we merge the two rightmost sibling leafs $\{\!\!\{Pos\ P,\ Pos\ Q\}\!\!\}$ and $\{\!\!\{Pos\ P\}\!\!\}$: the atom Q does not appear in one of the two formulas, so we can replace the node Q by a leaf containing the formula $\{\!\!\{Pos\ P\}\!\!\}$. We have not done any use of the rule.

We can merge the two other sibling leafs with formulas $\{\!\!\{Neg\ P,\ Neg\ Q\}\!\!\}$ and $\{\!\!\{Neg\ P,\ Pos\ Q,\ Pos\ Q\}\!\!\}$: the literal Q appears in both formulas. The first step is to remove the duplicate in the formula to get $\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}$. The associated transition is $(N, \emptyset) \Rightarrow_{\text{Res}} (N \cup \{\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}\}, \emptyset)$. Then we can apply the Res rule to $\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}$ and $\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}$ to get $\{\!\!\{Neg\ P,\ Neg\ P\}\!\!\}$: $(N \cup \{\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}\}, \emptyset) \Rightarrow_{\text{Res}} (N \cup \{\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}, \{\!\!\{Neg\ P,\ Neg\ P\}\!\!\}\}, \emptyset)$ (result in Figure 5c). We did not use any rule here.

We have reduced the number of variables that appear in the leaf. We can now merge the last two sibling leafs $\{\!\!\{Neg\ P,\ Neg\ P\}\!\!\}$ and $\{\!\!\{Pos\ P\}\!\!\}$: we first reduce the number of occurrences of *Neg P* and apply the Res rule: we get $\bot$ (Figure 5d). We have applied two rules of the calculus.

We have now finished to merge the sibling leafs and there is a single remaining leaf containing the formula $\bot$: this is what we wanted to have. More generally, at the end we have a formula φ such that the empty valuation is not a model: $[\,] \nvDash \varphi$. The only solution is that φ is (as here) $\bot$: we have deduced what we wanted to have.

In the previous example we have have skipped the conditions on *already-used*. To take care of the conditions, we use the following invariant:

Definition 10. • *For each pair (A, B) that has been already used, there is an atom P such that Pos P is in A and Neg P is in B.*

- *Either there is χ that subsumes the conclusion (i.e. χ implies the conclusion of the* Res *rule $A - \{\!\!\{Pos\ P\}\!\!\} + (B - \{\!\!\{Neg\ P\}\!\!\})$) or the latter is a tautology. any formula).*

The part of the invariant with the tautology is for the case that we want to apply the Res rule to the same two clauses only changing the literal. For example if we have $\{\!\!\{Pos\ P,\ Pos\ Q\}\!\!\}$ and $\{\!\!\{Neg\ P,\ Neg\ Q\}\!\!\}$, we can apply the Res rule with P or Q: in each case we have a tautology. In that case every possible conclusion is a tautology (including the previous conclusion). In the semantic tree, we cannot have this case since we are always working with conflicts: $I \not\models \varphi$ implies that I is not tautology. This shows also that re-applying the Res rule on the same two clauses, even when changing the variables, does not lead to any progress in the proof.

The part with the subsumption is not necessary here, since we do not remove any clause. However, it allows to use the conclusion where some duplicate have already been removed: if we want to re-apply the resolution rule on $\{\!\!\{Neg\ P,\ Neg\ Q\}\!\!\}$ and $\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}$, and $\{\!\!\{Neg\ P\}\!\!\}$ is already in the set of clauses, we can directly take the version without duplicates.

$\square$

This theorem shows completeness and soundness of the resolution calculus: this proof is one of the proofs of Weidenbach's book, we were able to simplify.

5.3.2 The Calculus with reduction Rules

The problem with the inference system described above is that new clauses are only added, while it is useful to delete some clauses like the factoring rule: we have both $\{\!\!\{Pos\ P,\ Pos\ P\}\!\!\} + C$ and $\{\!\!\{Pos\ P\}\!\!\} + C$, while the second is enough (since they are equivalent). More precisely the rules described in Section 5.2 are *inference* rules since new clauses are added, while we will describe *reduction* rules that are useful to have fewer clauses (see Figure 6).

$$[\![A \in N;\ A \subset B;\ B \in N]\!] \Longrightarrow \textit{simplify}\ N\ (N - \{B\})$$

(a) *Subsumption*: each model of A is also a model of B, so the latter can be removed of A. $\subset$ is the strict inclusion (to ensures that A and B are different).

$$A + \{\!\!\{Pos\ P\}\!\!\} + \{\!\!\{Neg\ P\}\!\!\} \in N \Longrightarrow \textit{simplify}\ N\ (N - \{A + \{\!\!\{Pos\ P\}\!\!\} + \{\!\!\{Neg\ P\}\!\!\}\})$$

(b) *Tautology deletion*, we remove a clause.

$$A + \{\!\!\{L\}\!\!\} + \{\!\!\{L\}\!\!\} \in N \Longrightarrow \textit{simplify}\ N\ (N - \{A + \{\!\!\{L\}\!\!\} + \{\!\!\{L\}\!\!\}\} \cup \{A + \{\!\!\{L\}\!\!\}\})$$

(c) *Condensation* is close to the Fact rule but removes the premise.

Figure 6: Reduction rule

The rules are applied using a strategy: simplify as much as possible, then apply Res or Fact: this is the transition of the transition system. This means that (except the first state), all the formulas are always simplified. As we are interested in what happens after the simplification and the rules, we will consider a $\textit{simp}^{+\downarrow}$ that simplifies as much as possible. Then we get the two rules:

$$\frac{\textit{simp}^{+\downarrow}\ N\ N'}{\textit{resolution}\ (N,\ \textit{already-used})\ (N',\ \textit{already-used})}$$

$$\frac{(N,\ \textit{already-used}) \Rightarrow_{\text{Res}} (N',\ \textit{already-used}') \qquad \textit{simplified}\ N \qquad \textit{simp}^{\downarrow}\ N'\ N''}{\textit{resolution}\ (N,\ \textit{already-used})\ (N'',\ \textit{already-used}')}$$

Using a predicate that fully simplify the formula make thing easier, since we can assume that at each step (except the very first), we can show that we are always in a state where everything is simplified. There are two predicates *simp*$^{+\downarrow}$ and *simp*$^{\downarrow}$: the difference is that *simp*$^{+\downarrow}$ must do one step (in *simp*$^{+\downarrow}$ $N\ N'\ N$ is different of N'), whereas *simp*$^{\downarrow}$ do an arbitrary number of steps including zero. We cannot have a rule of the form *simp*$^{\downarrow}$ $N\ N' \Longrightarrow$ *resolution* $(N,\ al)\ (N',\ al)$, otherwise we cannot prove termination: we could remain in the same state while doing steps (*resolution* $(N,\ al)\ (N,\ al)$). *simp*$^{\downarrow}$ simplifies our state and if nothing can be done, then it does nothing. It is the inference rule that change the state in the rule.

The proofs are close to the proofs presented in previous version, but we have to show that *simp*$^{\downarrow}$ terminates, because we want to use the result *after* the simplification. Now the *subsumes* part in the invariant of Definition 10 is important: we remove clauses that are either tautologies or are replaced by a simpler clause. The simplifications does not preserve a semantic tree, because the formulas on the leafs can change; but there are replaced by another equivalent formula.

Verified Theorem 11. *The resolution calculus with reduction rules terminates.*

Proof. The idea of the proof is that we cannot apply the Fact rule, since the clauses are always simplified. Then at each step of already used clauses is strictly increasing. As there is only a finite number of simplified clauses containing only the atoms of the set of clauses, we have termination. □

The proof of this theorem is one of the few that were broken in Weidenbach's book: the given argument was wrong. The proof tried to prove a better bound than ours (our upper bound is 3^n Res steps where n is the number of clauses).

The length of the full formalisation of resolution only is 1 200 lines of Isabelle. The resolution calculus is based on combining clauses. We will now present another algorithm that is based on case distinction of the values of the atoms.

6 The Davis-Putnam-Logemann-Loveland Calculus

The idea of the previous calculus was to apply rules on the set of clauses whose unsatisfiability we want to show, until we derive $\bot$. The idea now is to "try" values for propositional variables and when we have found a contradiction to backtrack on the decision and try the other choice. We will first describe the rules (Section 6.1), then give an example (Section 6.2). After that we will describe the Isabelle formalisation (Section 6.3), before describing a simple implementation (Section 6.4).

6.1 The Rules

The Davis-Putnam-Logemann-Loveland (DPLL) procedure is a procedure developed in 1962 [22, 23]. The idea is to build a sequence of literals that have been assigned: if we have to "try" a value we mark the corresponding literal with $^+$. After our choice, we try to find a contradiction and when we have found one, we backtrack on our last choice and choose the opposite value.

More formally we start with the pair $(\varepsilon; N)$: N is the set of clauses and initially no value has been defined and ε is the empty list. At the end we have either $(M; N)$ where $M \vDash N$ (meaning that we have found a model) or $(M; N)$ where $M \vDash \neg N$ without marked variables in M: N is unsatisfiable. By construction, M will not contain a contradiction (no atom appears more than once in M and especially it cannot appear both positively and negatively). Here are the rules:

- Propagate (Prop): if $C \vee L \in N$ and $M \vDash \neg C$ and the atom L has not yet been defined in M, then $(M; N) \Rightarrow_{\text{DPLL}} (L \cdot M; N)$. We do not have to backtrack on that decision, since we have taken the only possible decision for L.

  We write $|L| \notin_l |M|$ to mean that L has not been defined in M, i.e. neither L nor $-L$ is in M. Contrary to Weidenbach's presentation, the literal is consed on the left and not on the right: his cons operator appends the element to the list (i.e. of type *'a list ⇒ 'a ⇒ 'a list*). In Isabelle the cons operator adds an element at the beginning (i.e. of type *'a ⇒ 'a list ⇒ 'a list*). As Isabelle lists are a very developed with a lot of automation, we flipped the direction of the list M between the presentation in the book and in our files.

- Decide (Dec): if a literal L is undefined in M, we can define it: $(M; N) \Rightarrow_{\text{DPLL}} (L^+M; N)$. In that case we have to backtrack later on the decision because we chose arbitrary whether L or $\neg L$ is put in M, thus the mark $^+$.

- Backtrack (Back): starting from $(M_2L^+M_1; N)$, if we have found a contradiction i.e. $\exists D \in N, M \vDash \neg D$, then we take the opposite of our last choice (i.e. there is no marked variable in M_2, $\neg(\exists K, K^+ \in M)$): $(M_2L^+M_1; N) \Rightarrow_{\text{DPLL}} (\neg LM; N)$. As this is the only other case case for the value on L, no mark is needed.

6.2 Example

We apply our procedure on the set: $N = \{\{\!|P,\ Q|\!\},\ \{\!|\neg\ P,\ Q|\!\},\ \{\!|P,\ \neg\ Q|\!\},\ \{\!|\neg\ P,\ \neg\ Q,\ S|\!\},\ \{\!|\neg\ P,\ \neg\ Q,\ \neg\ S|\!\}\}$ (the example comes from [9]). The Back rule stands for application of the backtrack rule, Dec for decide and Prop for Propagate. One possible trace of transitions is

$([],\ N)$	$\Rightarrow_{\text{DPLL}}$	$([P^+],\ N)$	Dec since $\lvert P\rvert \notin_l \lvert[]\rvert$
	$\Rightarrow_{\text{DPLL}}$	$([Q,\ P^+],\ N)$	Prop since $\{\!\lvert\neg\ P,\ Q\rvert\!\} \in N$
	$\Rightarrow_{\text{DPLL}}$	$([S^+,\ Q,\ P^+],\ N)$	Dec since $\lvert S\rvert \notin_l \lvert[Q,\ P^+]\rvert$
	$\Rightarrow_{\text{DPLL}}$	$([\neg\ S,\ Q,\ P^+],\ N)$	Back since $[S^+,\ Q,\ P^+] \models as\ (\neg\ \{\!\lvert\neg\ P,\ \neg\ Q,\ \neg\ S\rvert\!\})$
	$\Rightarrow_{\text{DPLL}}$	$([\neg\ P],\ N)$	Back since $[\neg\ S,\ Q,\ P^+] \models as\ (\neg\ \{\!\lvert\neg\ P,\ \neg\ Q,\ S\rvert\!\})$
	$\Rightarrow_{\text{DPLL}}$	$([Q,\ \neg\ P],\ N)$	Prop since $\{\!\lvert P,\ Q\rvert\!\} \in N$

Now we have found a conflict: $\{\!|P,\ \neg\ Q|\!\} \in N$ but is false. We could stop here (we now know that the set of clauses N is unsatisfiable), but we can still preform some transitions:

$\Rightarrow_{\text{DPLL}}$	$([(\neg\ S)^+,\ Q,\ \neg\ P],\ N)$	Dec since $([Q,\ \neg\ P],\ N)$
$\Rightarrow_{\text{DPLL}}$	$([S,\ Q,\ \neg\ P],\ N)$	Back since $[(\neg\ S)^+,\ Q,\ \neg\ P] \models as\ (\neg\ \{\!\lvert P,\ \neg\ Q\rvert\!\})$

There is no remaining transition: every atom of the problem has a value and we cannot backtrack, since there is no marked variable. As the found valuation is not a model of N, N is unsatisfiable.

We can see here that sometimes we can apply rules but we already know whether the set of clauses is satisfiable. This kind of state is a final state (written *final-dpll-state* $(M,\ N)$), but not a termination state for the transition system. In practice, most implementations do no stop, but as we did here, propagate all the variables: the reason is that $M \models N$ is expensive to test: you have to test that every clause in N has M as model, so it is cheaper to verify to finish the propagation. Rule Propagate is not needed to show completeness since we can apply decide on the the opposite and then backtrack on the choice, but ignoring it can lead to exponentially longer solution. The strategy of ignoring Propagate is included in our transition system: we do not give any constrain on which rule to apply.

6.3 Isabelle Formalisation

We have formalised the rules described previously (described in Section 6.3.1). Then we will give some indications of the invariants needed for the proof (Section 6.3.2). When doing transitions, there are states that allows to conclude even if there are more possible transitions; these final states are described in Section 6.3.3. Then we show the theorem of correctness in Section 6.3.4

6.3.1 The Rules

We use a generalised version of the marked literals (to share definitions and lemmas between this section and the next section) with annotations on both marked and not marked literals:

datatype $('v,\ 'l,\ 'm)$ *marked-lit* =
Marked (*lit-of*: *'v literal*) (*level-of*: *'l*) | *Propagated* (*lit-of*: *'v literal*) *'m*

For DPLL the mark is a simple constant called *Level* for the level and *Proped* for the propagation. *lit-of* is the function that give the literal that is marked or propagated, while *lits-of* returns the literals in a list of marked or propagated literals. We will nevertheless write L^+ for *Marked L Level* and L for *Propagated L Proped*.

Now we can define a new version of $\models$: $(I \models a\ C) = (\textit{lits-of}\ I \models C)$. We translate the $\models a$ ("a" stands for annotated) into $\models$, using the function *lits-of*.

The rule of the calculus are:

- Propagate:

$$\frac{C + \{\!\!\{L\}\!\!\} \in N \qquad M \models as\ (\neg\ C) \qquad |L| \notin_l |M|}{(M,\ N) \Rightarrow_{\text{DPLL}} (L \cdot M,\ N)}$$

- Decide:

$$\frac{|L| \notin_l |M| \qquad |L| \in \textit{atms-of-m}\ N}{(M,\ N) \Rightarrow_{\text{DPLL}} (L^+ \cdot M,\ N)}$$

  Unlike Weidenbach's presentation, we have added the condition $|L| \in \textit{atms-of-m}\ N$. This (obvious) condition is needed for the soundness of the calculus.

- Backtrack: The distance between the paper version and the Isabelle version is the largest in this case, because there are implicit information.

$$\frac{\textit{backtrack-split}\ M = (M',\ L \cdot M) \qquad \textit{is-marked}\ L \qquad D \in N \qquad M \models as\ (\neg\ D)}{(M,\ N) \Rightarrow_{\text{DPLL}} (-\ \textit{lit-of}\ L \cdot M,\ N)}$$

backtrack-split $M = (M, L\ M')$ is a decomposition such that that $S = M\ @\ (L \cdot M')$. The marked literal is the first element of M' when such an element exists, otherwise $M' = []$. Using $(M', \textit{Marked } L \cdot M')$ instead of tuple $(M', \textit{Marked } L, M)$ allows a unified vision if there is no marked literal. Notice that *is-marked* L is redundant given the definition of *backtrack-split*, but we stay closer to the definition when adding the condition.

Every rule implies a progress: there is no explicit backtracking where we would go back to a previous state and change the decision taken there.

Now we have stated the definitions, we will give an overview of the proofs.

6.3.2 Invariants

Most invariants are written assuming that the initial state is $([], N)$, but it is easier to write the properties as invariants: it avoids doing induction on the possible transition and on the iteration of the transitions.

One of the very first properties to show is the following:

Verified Theorem 12. *If* $(M, N) \Rightarrow_{\mathrm{DPLL}} (M', N')$ *then* $N = N'$.

This is correct by the way the rules are stated, but we have nevertheless to write this property down and prove it by induction.

Textbook Theorem 13. *The sequence M will, by construction, neither contain duplicate nor complementary literals.*

We will define *no-dup* being the property that there is no duplicate, in the sense that no atom is defined twice. This property subsumes the duplicate freedom and the complementary freedom of the previous theorem.

Verified Theorem 13. *If* $S \Rightarrow_{\mathrm{DPLL}} S'$ *and no-dup* $(\textit{fst } S)$ *then no-dup* $(\textit{fst } S')$.

Proof. Isabelle is not fully convinced by the argument "by construction", but the proof is simply an induction on the possible transitions. Only the backtrack case needs a little more work in Isabelle, because of the splitting. ☐

When applying the rule Propagate, we do not have to backtrack on that decision, since we have taken the only possible decision for L. This leads to the following theorem:

Textbook Lemma 14. *et* $(M;\ N)$ *be a stated reached by the DPLL algorithm from the initial state* $(\varepsilon; N)$. *If* $M{=}M_{m+1}{L_m}^{+}\ldots{L_1}^{+}M_1$ *and all* M_i *have no decision literal then for all* $1 \leq i \leq m$ *it holds:* $N, M_1,\ldots, {L_i}^{+} \models M_{i+1}$.

In Isabelle it is a bit more complicated since writing the decomposition down is not as easy. *get-all-marked-decomposition* gives all the possible decomposition of the form $(M_i, L_i \cdot M)$ where no variable is marked in M. Then we want to show that for each of this decomposition N, M, L_i $\models M$. In Isabelle, the $\models$ is not overloaded, so it becomes:

$$\textit{unmark } (L_i \cdot M) \cup N \models\!\textit{ps unmark } M_i$$

unmark $(L_i \cdot M)$ converts all the marked literals from $L_i \cdot M$ to normal literals. More precisely they become clauses composed of a single literal, meaning that each literal have to be true. The Isabelle needs one more assumption to prove it: all the atoms in M have to be in the atoms of N. This condition is written *atm-of ‘ lit-of ‘ M* $\subseteq$ *atms-of-m N*: the operator ‘ means that

Verified Lemma 14. *If all-decomposition-implies N (get-all-marked-decomposition M) and one DPLL step* $(M, N) \Rightarrow_{\text{DPLL}} (M', N')$ *is done and the condition on the atoms hold atm-of ‘ lit-of ‘ M* $\subseteq$ *atms-of-m N, then all-decomposition-implies N′ (get-all-marked-decomposition M′).*

This conclusion can be generalised.

Textbook Lemma 15. *th-dpll-prop-gene* $(\varepsilon; N) \Rightarrow^{\star}_{DPLL} (M, N)$ *where* $M = M_{m+1} \cdot L^+_m \cdots L^+_1 \cdots M_1$ *and there is no decision literal in the* M_i*. Then:* $N, L^+_1, \ldots, L^+_i \vDash M_1, \ldots, M_{i+1}$.

Verified Lemma 15. *If one step* $(M, N) \Rightarrow_{\text{DPLL}} (M', N')$ *is done, every atom in M is also in N atm-of ‘ lit-of ‘ M* $\subseteq$ *atms-of-m N and all-decomposition-implies N (get-all-marked-decomposition M), then snd S′1* $\cup$ $\{\{\!|$*lit-of L*$|\!\}$ $\mid$ *is-marked L* $\wedge$ *L* $\in$ *fst S′1*$\}$ $\models$*ps unmark (fst S′1).*

Verified Lemma 16. *If M contains only propagated literals and there is D* $\in$ *N with M* $\models$*as* $(\neg$ *D) then N is unsatisfiable*

The Isabelle version needs some more assumptions, because of our presentation as invariants: the invariants have to be stated explicitly.

Verified Theorem 17. *If all-decomposition-implies N (get-all-marked-decomposition M) and D* $\in$ *N and M* $\models$*as* $(\neg$ *D) and* $\forall\, x{\in}M.\ \neg$ *is-marked x and atm-of ‘ lits-of M* $\subseteq$ *atms-of-m N then unsatisfiable N.*

This theorem shows that an important property: if we have found a contradiction while running the DPLL and no variable is marked, then the set of clauses is unsatisfiable. This is part of the proof of correctness.

6.3.3 Final States

Final states are states where can conclude: if M is a model of N, we know that N is satisfiable, even if M is not total. Otherwise if we have found a conflict and no variable is marked (i.e. we cannot backtrack), then N is unsatisfiable. The formal Isabelle definition is the following:

Definition 18. *final-dpll-state S = (fst S* $\models$*as snd S* $\vee$ $(\forall\, L{\in}$*fst S.* $\neg$ *is-marked L)* $\wedge$ $(\exists\, C{\in}$*snd S. fst S* $\models$*as* $(\neg$ *C))).*

This definition is linked to the termination of the final states of the rewrite system:

Verified Theorem 19. *If we are in state S and we cannot do any transition (i.e.* $\forall\, S'.\ \neg\ S \Rightarrow_{\text{DPLL}} S'$*), then we are in a final state: final-dpll-state S.*

As we have seen in the example of Section 6.2, the converse is not true: we can be in a final state, but there might be more rewrite steps to do. In practice, testing the satisfiabilty is too hard to be tested. If any of these conditions are met:

- we have found a contradiction and cannot backtrack;
- we have done every possible transition.

Theses conditions are cheap to verify in an implementation, contrary to the definition of a final states.

6.3.4 Correctness Theorems

There states where we know the satisfiability of the set of clauses before having done all the possible transitions, contrary to the resolution calculus where all inferences have to be done to conclude. However the termination of the rewrite system is enough to prove that we get in final state (Theorem 19): if we are lucky we get a final state early in the trace, but at latest when no more transition is possible, we are in a final state. That is why on one side the completeness theorem is expressed using the *final-dpll-state* predicate and the termination is proved one the transition system without the predicate.

The completeness and soundness theorem is the following:

Verified Theorem 20 (Completeness, Soundness)**.** *If* $([],\ N) \Rightarrow_{\mathrm{DPLL}}{}^{**}\ (M,\ N)$ *and final-dpll-state* $(M,\ N)$ *and finite* N *then* $(M \models\!as\ N) = satisfiable\ N$.

The termination (no more rewriting step can be done) can be proven using a well-founded order: the lexicographic order over natural numbers. If $M = L_n \cdots\ L_2 \cdot L_1 \cdot []$, then we define the following measure converting the state to a list:

$$\mu(M,\ N) = m_n \cdot \ldots \cdot\ m_2 \cdot m_1 \cdot \mathit{3} \cdot \ldots \cdot \mathit{3}$$

where m_i is 2 when L_i is annotated, *1* otherwise. and there are as many *1* as non-assigned variable in M. In Isabelle this becomes

$$dpll\text{-}mes\ M\ n = map\ (\lambda l.\ \mathsf{if}\ is\text{-}marked\ l\ \mathsf{then}\ \mathit{2}\ \mathsf{else}\ \mathit{1})\ (rev\ M)\ @\ replicate\ (n - |M|)\ \mathit{3}$$

where *map f l* applies f on every element of the list l.

The termination comes from the decreasing of the measure μ with respect to the lexicographic order. The lexicographic order with respect to order $\prec$ is *lex* $\prec$. To prove it, we need some of the invariants:

Verified Lemma 21. *If* $(M,\ N) \Rightarrow_{\mathrm{DPLL}} (M',\ N')$ *and atm-of ‘ lit-of ‘* $M \subseteq$ *atms-of-m* N *and no-dup* M *and finite* N *then* $(dpll\text{-}mes\ M'\ |atms\text{-}of\text{-}m\ N'|,\ dpll\text{-}mes\ M\ |atms\text{-}of\text{-}m\ N|) \in lex\ \{(a,\ b) \mid a < b\}$.

$|atms\text{-}of\text{-}m\ N|$ is the cardinal of set composed of the atoms in N. Although we consider $(M,\ N) \Rightarrow_{\mathrm{DPLL}} (M',\ N')$, the conclusion is the other way around with $(M',\ N')$ appearing before $(M,\ N)$ $(dpll\text{-}mes\ M'\ |atms\text{-}of\text{-}m\ N'|,\ dpll\text{-}mes\ M\ |atms\text{-}of\text{-}m\ N|) \in lex\ \{(a,\ b) \mid a < b\}$.

Using this key lemma, we can show the iteration of the transition is well-founded. To prove it, we have to ensure that the needed invariants (all together named *dpll-all-inv*) are correct. Using Isabelle well-foundedness predicate *wf*:

Verified Theorem 22 (Well-foundedness of DPLL)**.** *wf* $\{(S',\ S) \mid dpll\text{-}all\text{-}inv\ S \wedge S \Rightarrow_{\mathrm{DPLL}} S'\}$

Given the definition of *wf*, we have the next state is the first element of the pair: the relation $\{(0,\ 1),\ (1,\ 2),\ (2,\ 3),\ \ldots\}$ is well-founded whereas the relation $\{\ldots,\ (-3,\ -2),\ (-2,\ -1),\ (-1,\ 0)\}$ is not.

We can remove the invariant of the condition in the set and show that

Verified Theorem 23. *If finite N then wf* $\{(S',\ [],\ N) \mid ([],\ N) \Rightarrow_{\mathrm{DPLL}}{}^{++}\ S'\}$.

The length of the formalisation is five hundred lines of Isabelle code, with around the same length of libraries that defines the marked literals.

6.4 Implementation in Isabelle

In this section, we give a simple Isabelle implementation of a verified solver trying to prove the satisfiability of a set of clauses. We will first describe the function implementing the transitions (Section 6.4.1). After this, the details are specific to Isabelle, its **function** definition (Section 6.4.2). Then we export our verified Isabelle implementation in OCaml using the code generation (Section 6.4.3): the exported code is a verified solver in a real-world language.

6.4.1 Transition Step

We use the following strategy:

- we first try to find a unit clause, i.e. a clause composed of a single literal, using the function *find-first-unit-clause*;
- otherwise if there is no unit clause, we try to find a contradiction, using the predicate $\exists\, C \in N.\ Ms \models as\ (\neg\ multiset\text{-}of\ C)$;
- otherwise if there is no conflict, we select an unused variable (taking the first one when iterating over the formulas, *find-first-unused-var*);
- otherwise, the arguments are simply returned.

In our presentation we used sets, but in our program we will use lists. Moreover we will use integers *int* instead of type *'a*.

```
    definition DPLL-step :: int dpll-annoted-lits × int literal list list ⇒ int dpll-annoted-lits
× int literal list list  where
    DPLL-step = (λ(Ms, N).
      case find-first-unit-clause N Ms of
        Some L ⇒ (L · Ms, N))
      | None ⇒
        if ∃ C∈N. Ms ⊨as (¬ multiset-of C)
        then
          case backtrack-split Ms of
            (-, []) ⇒ (Ms, N) |
          | (-, L · M) ⇒ (− lit-of L · M, N)
        else
          case find-first-unused-var N (lits-of Ms) of
    $None \Rightarrow (Ms,\ N)$
  | $Some\ a \Rightarrow (a^{+} \cdot Ms,\ N)$
```

We do not have used any special data structure nor good strategy, since the implementation is only for presentation purpose.

When no step is possible, then the function *DPLL-step* returns its arguments. As we are using different types than the one described in our previous section, we define *toS Ms N* that converts the state $(Ms,\ N)$ from the type *int dpll-annoted-lits* $\times$ *int literal list list* to the type presentation we used in the previous section with sets. The correctness theorem associated with this definition is the following:

Verified Theorem 24. *If* $(Ms',\ N') = DPLL\text{-}step\ (Ms,\ N)$ *and* $(Ms,\ N) \neq (Ms',\ N')$ *then* $toS\ Ms\ N \Rightarrow_{\mathrm{DPLL}} toS\ Ms'\ N'$.

This lemma shows that *DPLL-step* does some subset of the possible transitions, but this is not enough: we can prove the same lemma for the identity function, since the condition $(Ms,\ N) \neq (Ms',\ N')$ is always false. We have to show that *DPLL-step* is doing a step until a final state is reached.

Verified Theorem 25. $(Ms,\ N) = DPLL\text{-}step\ (Ms,\ N) \Longrightarrow final\text{-}dpll\text{-}state\ (toS\ Ms\ N)$.

6.4.2 Combining and Termination

Using the previously defined *DPLL-step*, we would like to combine to write something like:

```
function DPLL-nt:: int dpll-annoted-lits ⇒ int literal list list
    ⇒ int dpll-annoted-lits × int literal list list where
DPLL-nt Ms N =
  (let (Ms', N') = DPLL-step (Ms, N) in
   if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-nt Ms' N)
```

But we cannot prove the termination, since the function *DPLL-nt* is not terminating (*nt* stands for non terminating): we have no control on the arguments $(Ms,\ N)$ so they can be in an a state where our invariants needed to prove termination do not hold.

A first way to define it, where we still have a total function, is to stop the execution whenever the invariant is false:

```
function DPLL-ci:: int dpll-annoted-lits ⇒ int literal list list
    ⇒ int dpll-annoted-lits × int literal list list where
DPLL-ci Ms N =
  (if ¬dpll-all-inv (toS Ms N)
  then (Ms, N)
  else
   let (Ms', N') = DPLL-step (Ms, N) in
   if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-ci Ms' N)
```

Whenever the invariant *dpll-all-inv* $(toS\ Ms\ N)$ is false, we stop the execution: *ci* in the function name stands for *check invariant*. Otherwise we use the order given by Theorem 22, but with the state conversion. Here is the beginning of the proof. We give the well-founded relation to use:

termination
  apply (*relation* $\{(S', S).\ (toS'\ S',\ toS'\ S) \in \{(S', S).\ dpll\text{-}all\text{-}inv\ S \wedge dpll\ S\ S'\}\}$)

Then we can show that *DPLL-ci* is really doing some steps of the transition systems:

Verified Theorem 26. *DPLL-ci Ms N* $= (Ms', N') \Longrightarrow toS\ Ms\ N \Rightarrow_{\mathrm{DPLL}}{}^{**}\ toS\ Ms'\ N.$

Proof. The proof is easy. If the invariant is false, then we do not do any step ($Ms = Ms'$ and $N = N'$) and we use the reflexivity of the relation. Otherwise, we have done a single step. □

We do not need any assumption about the invariant: whenever it is false, the conclusion comes from the fact that $op \Rightarrow_{\mathrm{DPLL}}{}^{**}$ is reflexive.

This version is not adapted to an implementation: we do not want to verify that the invariant holds at each step. A solution is to use **function** [24] such that termination is not required everywhere: then the function is only specified on its domain and theorems (like simplification rule) can only be applied if we are in the domain.

function (*domintros*) *DPLL-part*:: *int dpll-annoted-lits* ⇒ *int literal list list*
  ⇒ *int dpll-annoted-lits* × *int literal list list* **where**
DPLL-part Ms N =
 (*let* (*Ms′*, *N′*) = *DPLL-step* (*Ms*, *N*) *in*
  if (*Ms′*, *N′*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-part Ms′ N*)

The simplification are generated automatically by the **function** package. The *DPLL-part* encodes non termination: the domain is defined as all the points where the procedure terminates. The aim is to find conditions such that where are in the domain, as for example: *dpll-all-inv* $(toS'\ S) \Longrightarrow$ *DPLL-part-dom S*

If the function would not terminate, then the domain would be empty. For terminating functions, the condition on the domain is always true. The function *DPLL-part* is deeply linked to *DPLL-ci* when the invariant is verified:

Verified Theorem 27. *If dpll-all-inv* (*toS M N*) *then DPLL-part M N* = *DPLL-ci M N*.

This function is better suited for a code exportation since the invariant is not checked at each step (and the code exportation to verify is not invariant).

6.4.3 Code Generation

In the previous subsection, we implemented a concrete solver based on DPLL in Isabelle. To export the code in OCaml an obtain a verified solver, we use the *code-generation* tool [25]. It cannot handle partial function so we introduce a type that embeds the invariant. This follows the classical idiom when using the code generator. Isabelle has a mechanism to define new types as isomorphic to a subtype of an already defined type. Here our new type *dpll-state* is isomorphic to the :

typedef *dpll-state* = {(*M*::(*int*, *dpll-marked-level*, *dpll-mark*) *marked-lit list*, *N*::*int literal list list*). *dpll-all-inv* (*toS M N*)}
  morphisms *rough-state-of state-of*

Here our new type *dpll-state* is isomorphic to the elements of type (*int*, *dpll-marked-level*, *dpll-mark*) *marked-lit list* × *int literal list list* such that the invariant *dpll-all-inv* is true.

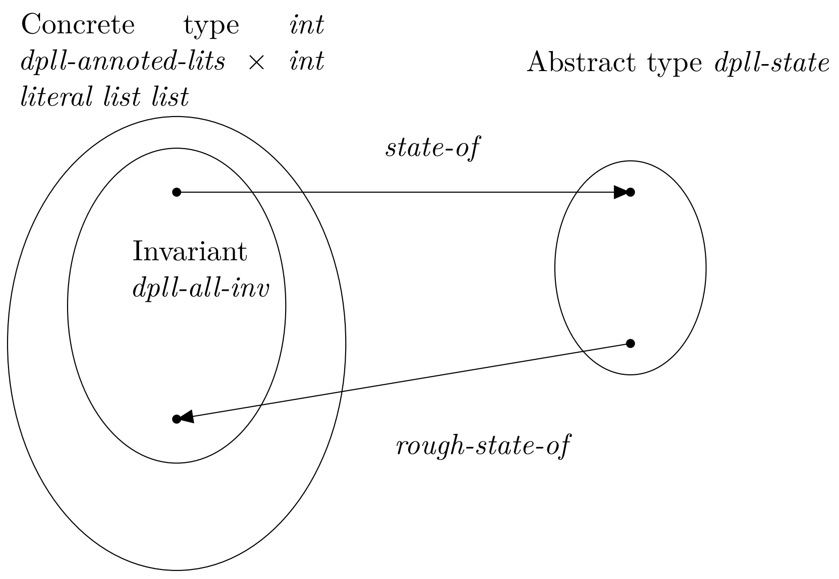

Concrete type *int dpll-annoted-lits* × *int literal list list*
Abstract type *dpll-state*
state-of
Invariant *dpll-all-inv*
rough-state-of

Figure 7: Morphisms between the abstract type and the concrete type with the invariant: the invariant folds only on a subset of the concrete type, and only this part can be linked to the abstract type.

Now we have defined the type, we can use it and define *DPLL-tot*: *DPLL-tot S* = (let *S'* = *DPLL-step' S* in if *S'* = *S* then *S* else *DPLL-tot S'*) where *DPLL-step'* :: *dpll-state* ⇒ *dpll-state* is *DPLL-step*, but converts its arguments from the abstract type forth to use *DPLL-step* and back after the step. This definition of *DPLL-tot* allows to prove termination.

The definition allows to prove the same properties than the other definition, especially the correctness of the transformation.

Verified Theorem 28 (Correctness of *DPLL-tot*)**.** *If we have evaluated our function, i.e. rough-state-of (DPLL-tot (state-of ([], N))) = (M, N') and (M', N'') = toS' (M, N'), then the following equivalence holds: satisfiable N''⟷ M' ⊨as N''.*

We can now use Isabelle *export-code*, that allows to generate code from our *DPLL-tot* definition. We have to define a constructor *Con* going from the concrete type to the abstract type: *Con xs* = *state-of* (if *dpll-all-inv* (*toS* (*fst xs*) (*snd xs*)) then *xs* else ([], [])). Given the conditions, we have proven the termination (since *DPLL-tot* terminates) and the correctness.

Here is an extract of the generated code in OCaml:

```
let rec rough_state_of (Con x) = x;;
let rec dPLL_stepa s = Con (dPLL_step (rough_state_of s));;
let rec dPLL4 s =
   let sa = dPLL_stepa s in
      (if equal_dpll_state sa s then s else dPLL4 sa);;
```

There is no invariant check, thanks to our type definition. The result is a proven implementation, if the code generation, the Isabelle kernel and the OCaml compiler are trusted. There is not invariant check.

We have presented the DPLL procedure and a simple implementation. We will now present an evolution of this procedure, that is faster in practice.

7 The Conflict-Driven Clause Learning Procedure

Conflict-driven clause learning (CDCL) is an extension of the DPLL procedure, based on the idea that backtracking more than one level is more efficient. We will first describe the rules (Suction 7.1), then we will show an example (Section 7.2) before giving more details about the proof and the strategy (Section 7.3). In this section we assume that the clauses are without duplicate literals.

7.1 Rules

A state is a tuple $(M; N; U; k; C)$ where M is the model we are working on: it is as before a sequence of marked literals. The marks are not the same as before: decision literals (marked with $+$ in the previous section) are now marked with a natural number; propagation literals are marked with a clause (the number of variables to backtrack on). N is the set of clauses we are considering. k is an integer representing the level of backtracking. C is a non-empty clause (for backtracking states) or $\top$ or $\bot$ ($\bot$ means that the search of a model has not been succesful so far) and U is a set of clauses to keep the clause that previously caused a contradiction.

The level of an atom ℓ in M is j when $Pos \ell \in M_j$ or $Neg \ell \in M_j$ and $M = M_n \cdot L^n \cdots M_j L^j \cdots M_1 \cdot L^1 \cdot M_0$, or 0 if ℓ is not in M. The level of the literal L is the level of the atom of L. The level of a clause is the maximum of the levels of the literals.

We start with $(\varepsilon; N; \emptyset; 0; \top)$ and we want to reach a final state: either $(M; N; U, k; \top)$ where $M \models N$, or $(M; N; U, k; \bot)$ meaning that N is unsatisfiable. During the search, an intermediate proof step is characterised by $M \not\models N$. Here are the rules:

Propagate (Prop): as in DPLL, we use our knowledge. If $M \models \neg C$ and $C \vee L \in N \cup U$ where L has not yet been defined, then $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{C \vee L}; N; U; k; \top)$.

Decide (Dec): we can also determine the value of a literal L, if it is undefined in M:

$$(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{k+1}; N; U; k+1; \top)$$

The level of backtracking is increased, since we have decided a new value to backtrack later on.

Conflict (Con): if there is a conflict in our state $(M; N; U, k; \top)$, i.e. $D \in N \cup U$ and $M \models \neg D$, then we "mark" the conflict: $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$.

Skip (Skip): if we are in a situation where we are backtracking ($D \notin \{\bot, \top\}$) and $\neg L$ does not occur in D (i.e. does not participate to the conflict), then we can remove the assignation of the literal, $(ML^{C \vee L}; N; U; k; D) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$.

Resolve (Resolve): if D contains a literal of level k or $k = 0$:

$$(ML^{C \vee L}; N; U; k; D \vee \neg L) \Rightarrow_{\text{CDCL}} (M; N; U; k; D \vee C)$$

The idea of the rule is to go back in to our previous state. Imagine we have the state $(ML^{C \vee L}; N; U; k; \bot \vee \neg L)$: we have chosen L, but get a conflict after. We change our decision, but as our decision was necessary (see rule Prop), the conflict must be in C: $(M; N; U; k; \bot \vee C)$.

Backtrack (Back): one important rule is the backtracking:

$$(M_1 K^{i+1} M_2; N; U; k; D \vee L) \Rightarrow_{\text{CDCL}} (M_1 L^{D \vee L}; N; U \cup \{D \vee L\}; i; \top)$$

given that the literal L is of maximal level k and D is of level i where $i < k$ (i.e. each literal in D is of maximum level i). The CDCL backtrack rule generalises the DPLL backtrack rule: i is the level we jump back, while in DPLL we can only go back one level.

Restart (Restart): We restart our actual partial valuation M, but not the learnt clauses. This allows to restart and maybe find a conflict faster thanks to the learnt clauses. This rule is heuristically used in practice and can obviously lead to non-termination:

$$(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (\varepsilon; N; U; k; \top)$$

provided that $M \models as\ N$ is false.

Forget (Forget): We remove one of the learnt clause:

$$(M; N; U \cup \{C\}; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; \top)$$

provided that $M \models as\ N$ is false.

When no rule can be applied anymore, then either C is $\top$ and N is satisfiable and $M \models as\ R$, or C is $\bot$ and N is unsatisfiable. Notice that in both cases we cannot apply Restart and Forget: when C is $\top$, then $\neg\ M \models as\ N$ does not hold, and when C is $\bot$, then C is not equal to $\top$. Thus these states are real termination states.

7.2 Example

We use *C-True* instead of $\top$ and *C-Clause D* instead of D when D is different of $\top$. We apply our procedure on the set $N = \{\{\!\!\{Pos\ P,\ Pos\ Q\}\!\!\},\ \{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\},\ \{\!\!\{Pos\ P,\ Neg\ Q\}\!\!\},\ \{\!\!\{Neg\ P,\ Neg\ Q,\ Pos\ S\}\!\!\},\ \{\!\!\{Neg\ P,\ Neg\ Q,\ Neg\ S\}\!\!\}\}$.

$$\begin{aligned}
([],\ N,\ \emptyset,\ 0,\ \textit{C-True}) &\Rightarrow_{\text{CDCL}} ([(Pos\ S)^{1}],\ N,\ \emptyset,\ 1,\ \textit{C-True})\ (\mathsf{Dec})\\
&\Rightarrow_{\text{CDCL}} ([(Neg\ P)^{2},\ (Pos\ S)^{1}],\ N,\ \emptyset,\ 2,\ \textit{C-True})\ (\mathsf{Dec})\\
&\Rightarrow_{\text{CDCL}} ([(Pos\ Q)^{\{\!\!\{Pos\ P,\ Pos\ Q\}\!\!\}},\ (Neg\ P)^{2},\ (Pos\ S)^{1}],\ N,\ \emptyset,\ 2,\\
&\qquad \textit{C-True})\ (\mathsf{Prop})\\
&\Rightarrow_{\text{CDCL}} ([(Pos\ Q)^{\{\!\!\{Pos\ P,\ Pos\ Q\}\!\!\}},\ (Neg\ P)^{2},\ (Pos\ S)^{1}],\ N,\ \emptyset,\ 2,\\
&\qquad \textit{C-Clause}\ \{\!\!\{Pos\ P,\ Neg\ Q\}\!\!\})\ (\mathsf{Res})\\
&\Rightarrow_{\text{CDCL}} ([(Neg\ P)^{2},\ (Pos\ S)^{1}],\ N,\ \emptyset,\ 2,\ \textit{C-Clause}\ \{\!\!\{Pos\ P\}\!\!\})\\
&\qquad (\mathsf{Back})\\
&\Rightarrow_{\text{CDCL}} ([(Pos\ P)^{\{\!\!\{Pos\ P\}\!\!\}},\ (Pos\ S)^{1}],\ N,\ \{\!\!\{Pos\ P\}\!\!\},\ 0,\ \textit{C-True})\\
&\qquad (\mathsf{Back})
\end{aligned}$$

The result of Resolve rule is $\{\!\!\{Pos\ P,\ Pos\ P\}\!\!\}$. We must remove the duplicates; otherwise the conditions to apply the backtrack rule are not verified: $\{\!\!\{Pos\ P,\ Pos\ P\}\!\!\}$ is of the form $D + \{\!\!\{Pos\ P\}\!\!\}$, but D is of level 2 as $Pos\ P$ and D must be of level strictly less than P.

$$\begin{aligned}
&\Rightarrow_{\text{CDCL}} && ([(Pos\ Q)^{\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}},\ (Pos\ P)^{\{\!\!\{Pos\ P\}\!\!\}},\ (Pos\ S)^{1}],\\
& && N,\ \{\!\!\{Pos\ P\}\!\!\},\ 0,\ C\text{-}True)\ (\mathsf{Prop})\\
&\Rightarrow_{\text{CDCL}} && ([(Pos\ Q)^{\{\!\!\{Neg\ P,\ Pos\ Q\}\!\!\}},\ (Pos\ P)^{\{\!\!\{Pos\ P\}\!\!\}},\ (Pos\ S)^{1}],\\
& && N,\ \{\!\!\{Pos\ P\}\!\!\},\ 0,\ C\text{-}Clause\ \{\!\!\{Pos\ P,\ Neg\ Q\}\!\!\})\ (\mathsf{Prop})\\
&\Rightarrow_{\text{CDCL}} && ([(Pos\ P)^{\{\!\!\{Pos\ P\}\!\!\}},\ (Pos\ S)^{1}],\ N,\ \{\!\!\{Pos\ P\}\!\!\},\ 0,\ C\text{-}Clause\\
& && \{\!\!\{Neg\ P\}\!\!\})\ (\mathsf{Res})\\
&\Rightarrow_{\text{CDCL}} && ([(Pos\ S)^{1}],\ N,\ \{\!\!\{Pos\ P\}\!\!\},\ 0,\ C\text{-}Clause\ \bot)\ (\mathsf{Res})
\end{aligned}$$

The last rule simply consists in applying Resolution. At the end we get $C\text{-}Clause\ \bot$: N is not satisfiable.

7.3 Isabelle Formalisation

We first present the rules in Isabelle (Section 7.3.1), then we show some invariants (Section 7.3.2). If we do not restrict the rules, there is no termination: to prove it, we use a strategy (Section 7.3.3).

7.3.1 Rules

CDCL is an extension of DPLL and some types are shared between both approaches:

datatype (*'v*, *'level*, *'mark*) *marked-lit* =
  is-marked: *Marked* (*lit-of*: *'v literal*) (*level-of*: *'level*) |
  Propagated (*lit-of*: *'v literal*) (*mark-of*: *'mark*)

Re-using the definition allows a to share some lemmas between DPLL and CDCL. The conflicting clause is either $\top$ or a clause:

datatype *'a conflicting-clause* = *C-True* | *C-Clause 'a*

The separation between N and the learnt clauses in U is only to clarify the presentation: the learnt clauses are consequences of N, thus propagating can be done based on a clause being either in N or in U.

$$\frac{S = (M,\ N,\ U,\ k,\ C\text{-}True) \qquad C + \{\!\!\{L\}\!\!\} \in N \cup U \qquad M \models as\ CNot\ C \qquad |L| \notin_l |fst\ S|}{S \Rightarrow_{\text{CDCL}} (L^{C + \{\!\!\{L\}\!\!\}} \cdot M,\ N,\ U,\ k,\ C\text{-}True)}\ \mathsf{Prop}$$

$$\frac{S = (M,\ N,\ U,\ k,\ C\text{-}True) \qquad |L| \notin_l |M| \qquad |L| \in atms\text{-}of\text{-}m\ N}{S \Rightarrow_{\text{CDCL}} (L^{k + 1} \cdot M,\ N,\ U,\ k + 1,\ C\text{-}True)}\ \mathsf{Dec}$$

When a conflict is found we update the state:

$$\frac{S = (M,\ N,\ U,\ k,\ C\text{-}True) \qquad L \in N \cup U \qquad M \models as\ CNot\ L}{S \Rightarrow_{\text{CDCL}} (M,\ N,\ U,\ k,\ C\text{-}Clause\ L)}\ \mathsf{Conf}$$

$$\frac{S = (L^{C} \cdot M,\ N,\ U,\ k,\ \textit{C-Clause}\ D) \qquad -L \notin D \qquad D \neq \bot}{S \Rightarrow_{\mathrm{CDCL}} (M,\ N,\ U,\ k,\ \textit{C-Clause}\ D)}\textsf{Skip}$$

We can resolve a conflict: when we have done a case distinction before and found a conflict, we can combine both information:

$$\frac{\begin{array}{c} S = (L^{C + \{\!|L|\!\}} \cdot M,\ N,\ U,\ k,\ \textit{C-Clause}\ (D + \{\!|-L|\!\})) \\ \textit{get-maximum-level}\ D\ (L^{C + \{\!|L|\!\}} \cdot M) = k \vee k = 0 \end{array}}{S \Rightarrow_{\mathrm{CDCL}} (M,\ N,\ U,\ k,\ \textit{C-Clause}\ (\textit{remdups-mset}\ (D + C)))}\textsf{Res}$$

This is an important invariant of the section: we have to ensure that there is no duplicate in the formulas. To remove them, we use the function *remdups-mset*.

The backtrack rule is a generalisation of the backtrack rule of DPLL. To backtrack several levels, we introduced a function *get-all-marked-decomposition* that returns a list of all decomposition of of the form $M_1 \cdot L^k \cdot M_2$. This function is linked to the *backtrack-split* we used for DPLL by the following theorem: $\textit{backtrack-split}\ S = (M,\ L \cdot M') \Longrightarrow \textit{hd}\ (\textit{get-all-marked-decomposition}\ S) = (L \cdot M',\ M)$.

$$\frac{\begin{array}{c} S = (M,\ N,\ U,\ k,\ \textit{C-Clause}\ (D + \{\!|L|\!\})) \\ (K^{i\,+\,1} \cdot M_1,\ M_2) \in \textit{get-all-marked-decomposition}\ M \qquad \textit{get-level}\ L\ M = k \\ \textit{get-level}\ L\ M = \textit{get-maximum-level}\ (D + \{\!|L|\!\})\ M \qquad \textit{get-maximum-level}\ D\ M = i \end{array}}{S \Rightarrow_{\mathrm{CDCL}} (L^{D + \{\!|L|\!\}} \cdot M_1,\ N,\ U \cup \{D + \{\!|L|\!\}\},\ i,\ \textit{C-True})}\textsf{Back}$$

Remark that contrary to DPLL, we are not taking the negation of literal K in the new state, but a different literal L.

We have separated Restart and Forget from the other rules:

$$\frac{S = (M1,\ N1,\ U1 \cup \{C1\},\ k1,\ \textit{C-True}) \qquad \neg\ M1 \models\!\textit{as}\ N1}{S \Rightarrow_{\mathrm{CDCL}} ([],\ N1,\ U1,\ 0,\ \textit{C-True})}\textsf{Forget}$$

$$\frac{S = (M1,\ N1,\ U1,\ k1,\ \textit{C-True}) \qquad \neg\ M1 \models\!\textit{as}\ N1}{S \Rightarrow_{\mathrm{CDCL}} ([],\ N1,\ U1,\ 0,\ \textit{C-True})}\textsf{Restart}$$

7.3.2 Invariants without Strategy

There are a few theorems that can be proved without constrain on the rules, then in the next section we introduce a strategy, where the rules Restart and Forget are not applied.

When we use the previous rule without any strategy, there a few propositions that we can show: firstly that the that the levels on the literals are sorted from the backtracking level k to one. This is obvious given the construction, but has to be shown.

The learnt clauses are entailed by the clauses. To prove so we need the three following invariants (collectively called *cdcl-learnt-clause*):

- the learnt clauses are entailed by N: $N \models\!\textit{ps}\ U$, but $M \models\!\textit{as}\ \textit{CNot}\ C$;
- whenever the conflicting part is not true, then $N \models\!\textit{p}\ C$;

- for every mark, it is entailed by the clauses: $N \models ps$ *get-all-mark-of-propagated M*.

The proof that the learnt clauses are entailed by the set of clauses is very sketchy in Weidenbach's book and I could not understand the link between some of the arguments of in the proof and the argument. We can show the same property about propagated variables as we did for DPLL (Lemma 14):

Verified Theorem 29. *If:*

- *one step is done, i.e.* $S \Rightarrow_{\text{CDCL}} S'$*;*
- *our property holds in S: (all-decomposition-implies (clauses S) (get-all-marked-decomposition (fst S));*
- *our 3 previous invariants are true cdcl-learned-clause S;*
- *some properties about the length and the numbering are correct (i.e. the levels are [k... 1]);*
- *the defined literal are in N.*

then all-decomposition-implies (clauses S') (get-all-marked-decomposition (fst S')).

The proof is significantly more complicated than the proof of DPLL and this needs all the invariants that are given as assumption here. This theorem holds independently of the strategy of application of the rules and is enough to show the counterpart of lemma 2.9.2.

Verified Lemma 30. $([\,]; N) \Rightarrow^{\star}_{CDCL} (M, N)$ *where* $M = M_{m+1} \cdot L^{+}_{m} \cdots L^{+}_{1} \cdots M_1$ *and there is no decision literal in the* M_i*. Then* $N, L^{+}_{1}, \ldots, L^{+}_{i} \models M_1, \ldots, M_{i+1}$.

Although this properties are very general, we need some more specific constrains on how to apply the rules to prove termination for example: an infinite application of restarts is possible, but incompatible with termination.

7.3.3 Invariants with Strategy

Weidenbach's book [9] presents the strategy we will use, called *reasonable*: the rule Conf is preferred over Prop, and both are preferred over all other and we do not apply restart and forget. In an inductive predicate, there is no order between the different rules. So we create two inductive definitions: one for Prop and Conf *cdcl-cp* and one for the other rule *cdcl-o* (Back, Dec, Res). There is a strategy in *cdcl-cp*: Conf is preferred overProp. To do so, the rule is stated such that *propagate S S′* is applied, only if we cannot do any step with a Conf rule, i.e. *no-step conflict S*. To ease the proof, we apply the rule Conf after applying Prop if possible. This allows to be sure that there is no possible conflict after each application of *cdcl-cp* or it has been selected.

cdcl-cp is defined by:

- if there is conflict, we apply it:

$$\frac{\textit{conflict } S\ S'}{\textit{cdcl-cp } S\ S'}$$

- if there is no conflict *no-step conflict* S, then we do a propagation *propagate* S S' and the conflict *conflict* S' S'' after propagation.

$$\frac{\textit{propagate}\ S\ S' \qquad \textit{no-step conflict}\ S \qquad \textit{conflict}\ S'\ S''}{\textit{cdcl-cp}\ S\ S''}$$

- if there is no conflict *no-step conflict* S, then we do a propagation *propagate* S S' and there is no conflict *no-step conflict* S'.

$$\frac{\textit{propagate}\ S\ S' \qquad \textit{no-step conflict}\ S \qquad \textit{no-step conflict}\ S'}{\textit{cdcl-cp}\ S\ S'}$$

This presentation allows to prove that after a *cdcl-cp* step, then we have no conflict: ($\textit{conflicting}\ S = \textit{C-True} \longrightarrow (\forall D \in \textit{clauses}\ S \cup \textit{learned-clauses}\ S.\ \neg \textit{fst}\ S \models\!\textit{as}\ \textit{CNot}\ D)$).

Then we can define the full strategy:

- if we can apply *cdcl-cp*, we apply it as long as possible (more than once, thus the $^{+\downarrow}$):

$$\frac{\textit{cdcl-cp}^{+\downarrow}\ S\ S'}{\textit{cdcl-s}\ S\ S'}$$

- if we cannot apply either propagate (*no-step propagate* S) nor conflict (*no-step conflict* S), then we can apply another rule *cdcl-o* S S'. As before when then apply the *cdcl-cp* as long as possible, possibly zero times, using $^{\downarrow}$.

$$\frac{\textit{cdcl-o}\ S\ S' \qquad \textit{no-step propagate}\ S \qquad \textit{no-step conflict}\ S \qquad \textit{cdcl-cp}^{\downarrow}\ S'\ S''}{\textit{cdcl-s}\ S\ S''}$$

One of the important invariants is the following:

$$\forall D.\ \textit{conflicting}\ S' = \textit{C-Clause}\ D \longrightarrow D \neq \bot \longrightarrow$$
$$(\exists L.\ L \in D \wedge \textit{get-level}\ L\ (\textit{fst}\ S') = \textit{backtrack-level}\ S')$$

It states that whenever we have a conflict that is not $\bot$, then there is a literal of level the backtracking level. It explains why the backtracking conditions are not too restrictive: we can can remove literals until the marked variable, i.e. going from $L_1^{C_1} \cdot L_2^{C_2} \cdot \ldots \cdot L_n^{C_n} \cdot K^{i+1} \cdot M$ to $K^{i+1} \cdot M$ using the Skip and Res rules. Then we can apply the backtrack rule, since there is no duplicate and $K \in C$ we do not get stuck in this case. This important invariant does not appear in the proof in Weidenbach's book.

We call *S0-cdcl* N the initial state: it is an initial state only if N is finite (*finite* N) and has no duplicate (*no-dup-mset-set* N). The termination theorem is the following (the fact that we cannot do any more step is included in the definition of $^{+\downarrow}$).

Verified Theorem 31 (CDCL final states)**.** *If* $\textit{cdcl-s}^{\downarrow}$ $(\textit{S0-cdcl}\ N)$ S' *and* *no-dup-mset-set* N *and* *finite* N *then* $\textit{conflicting}\ S' = \textit{C-Clause}\ \bot \wedge \textit{unsatisfiable}\ (\textit{clauses}\ S') \vee \textit{conflicting}\ S' = \textit{C-True} \wedge \textit{fst}\ S' \models\!\textit{as}\ \textit{clauses}\ S'$.

This theorem can be verified with the Restart rule, but to prove termination we have to work without this rule. The termination car be shown with a measure:

```
    case C of
        C-True ⇒ [3 ^|atms-of-m N| − |U|, 1, |atms-of-m N| − length M]
      | C-Clause - ⇒ [3 ^|atms-of-m N| − card U, 0, length M]
```

We are using a list (and not a pair), since the lexicographic order in Isabelle is defined for lists. The decreasing itself is independent of the strategy, but there is one key property that depends on the strategy and has not been formalised in this work: the learnt clauses have not been learnt before. The proof depends on the formalisation of the superposition calculus (ordered resolution as presented in 5) to show it and on the link between CDCL and superposition. In Weidenbach's book [9], the superposition calculus has been already introduced before CDCL. The idea of superposition is to restrain the possible inference of resolution: the result of the inferences is not redundant with respect to our knowledge.

The length of the development is 2 500 lines of code: it is nearly five time longer that the DPLL formalisation (compare the 8 rules of CDCL to the 3 rules of DPLL). There are around five hundred lines of code of shared libraries between CDCL and DPLL to define the marked variables and various lemmas about them.

7.4 Formalisation of Modern SAT Solvers

While the resolution formalisation (Section 5) was not an efficient solver (its purpose was the completeness theorem, not efficiency), Marić [26] has developed a concrete CDCL prover and has verified it in Isabelle/HOL (the code can be found in the Archive of Formal Proofs [27]).

It provides an implementation in Haskell (thanks to Isabelle/HOL's code generator). The algorithm is implemented in a side-effect-free manner, and is less efficient than a C state-of-the-art prover, but it is proven that it terminates and that it finds a proof. There are two ways of trusting a prover:

(1) proving the program itself: it can be very hard for state-of-the-art algorithms, but allows a full trust into the prover (e.g. if you have proven completeness, then a result will always be found); checking that a model is correct is very easy. The other direction is the difficult one: if no conflict has been found, is the problem really unsatisfiable?

(2) extending a SAT solver so that it outputs a proof of the contradiction. Then an external proven verifier can be used to certify the output. It is often simpler to prove, but this approach has some drawbacks, because producing proofs causes an overhead, but allows implementing state-of-the art algorithm without having to prove it. Notice that this means that we can get *nothing*: if the prover always produces a wrong output, you only know that it is wrong. This approach does not really allow to prove completeness, since we have no properties on the SAT solver itself.

The first approach has been used by Marić, allowing someone to use a proven solver (meaning that if a result can be found, it will be found and if a result is found, it is correct). He has proven even more properties like termination, unlike Oe *et al.* in [28]: they have proven in Guru the correctness of a quite efficient implementation in C (using integer overflow for example, contrary to Marić), but no termination.

8 Conclusion

We have presented here the formalisation of the normalisation, the resolution calculus, the DPLL and CDCL procedures in Isabelle/HOL. This presentation as a transition systen allows to study easily changes in the definitions and the conditions on the transition, since Isabelle will tell exactly where it is not able to redo the proof.

We have done a full Isabelle developpement without **sorry** (the keyword in Isabelle to admit a lemma): our proof are complete and we have not used any axiom. We have shown soundness and completeness of the tree calculi and have produced a simple verified prover. This prover could be integrated to Isabelle as a new proof method. The actual method [29] replays the proof through the Isabelle kernel. We would not need that, since we have proved the correctness of our solver. During our developement, we have contributed to improve the library of multisets in Isabelle.

As a future work, formalising superposition to prove the missing assumption for the termination of CDCL. The superposition calculus (an extension of the resolution calculus) is also used (with different rules) for first-order logic in automated provers: it would be interesting to formalise it. With enough automation this could interest researches working on various fragments of first-order logic, to have more confidence in the proofs.

References

[1] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[2] Stephan Schulz. "System Description: E 1.8". English. In: *Logic for Programming, Artificial Intelligence, and Reasoning.* Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 735–743. ISBN: 978-3-642-45220-8. DOI: 10.1007/978-3-642-45221-5_49. URL: http://dx.doi.org/10.1007/978-3-642-45221-5_49.

[3] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. "SPASS Version 3.5". English. In: *Automated Deduction – CADE-22.* Ed. by RenateA. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 140–145. ISBN: 978-3-642-02958-5. DOI: 10.1007/978-3-642-02959-2_10. URL: http://dx.doi.org/10.1007/978-3-642-02959-2_10.

[4] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". English. In: *Computer Aided Verification.* Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–35. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_1. URL: http://dx.doi.org/10.1007/978-3-642-39799-8_1.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Proceedings of the* 23rd *International Conference on Computer Aided Verification (CAV '11).* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Snowbird, Utah. Springer, July 2011, pp. 171–177. URL: http://www.cs.nyu.edu/~barrett/pubs/BCD+11.pdf.

[6] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C.R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[7] Georges Gonthier. *Formal Proof – The Four-Color Theorem*. 2008.

[8] Thomas C. Hales. “A Proof of the Kepler Conjecture”. English. In: *Annals of Mathematics*. Second Series 162.3 (2005), ISSN: 0003486X. URL: http://www.jstor.org/stable/20159940.

[9] Christoph Weidenbach. “Automated Reasoning – The Art of Generic Problem Solving”. To appear, chapter 2, version as of 26. December 2014.

[10] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. “A Formally Verified Proof of the Prime Number Theorem”. In: *ACM Trans. Comput. Logic* 9.1 (Dec. 2007). ISSN: 1529-3785. DOI: 10.1145/1297658.1297660.

[11] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. “Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level”. In: *SIGOPS Oper. Syst. Rev.* 41.4 (July 2007), pp. 3–11. ISSN: 0163-5980. DOI: 10.1145/1278901.1278904.

[12] Martin Strecker. “Formal Verification of a Java Compiler in Isabelle”. English. In: *Automated Deduction—CADE-18*. Ed. by Andrei Voronkov. Vol. 2392. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 63–77. ISBN: 978-3-540-43931-8. DOI: 10.1007/3-540-45620-1_5. URL: http://dx.doi.org/10.1007/3-540-45620-1_5.

[13] Gerwin Klein and Tobias Nipkow. “A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler”. In: *ACM Trans. Program. Lang. Syst.* 28.4 (July 2006), pp. 619–695. ISSN: 0164-0925. DOI: 10.1145/1146809.1146811.

[14] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[15] Robin Milner. *Models of LCF*. Tech. rep. DTIC Document, 1973.

[16] Makarius Wenzel. “Isabelle/jEdit – A Prover IDE within the PIDE Framework”. English. In: *Intelligent Computer Mathematics*. Ed. by Johan Jeuring, JohnA. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge. Vol. 7362. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 468–471. ISBN: 978-3-642-31373-8. DOI: 10.1007/978-3-642-31374-5_38. URL: http://dx.doi.org/10.1007/978-3-642-31374-5_38.

[17] Mike Gordon. “Proof, Language, and Interaction”. In: ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000. Chap. From LCF to HOL: A Short History, pp. 169–185. ISBN: 0-262-16188-5.

[18] Lawrence C. Paulson and J. C. Blanchette. “Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers”. In: *IWIL-2010*. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC. EasyChair, 2012, pp. 1–11.

[19] Gerwin Klein, Tobias Nipkow, and Larry Paulson. *The archive of formal proofs*. 2010.

[20] René Thiemann and Christian Sternagel. "Certification of Termination Proofs Using CeTA". English. In: *Theorem Proving in Higher Order Logics.* Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 452–468. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_31. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_31.

[21] Markus Wenzel. "Isar — A Generic Interpretative Approach to Readable Formal Proof Documents". English. In: *Theorem Proving in Higher Order Logics.* Ed. by Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin. Vol. 1690. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 167–183. ISBN: 978-3-540-66463-5. DOI: 10.1007/3-540-48256-3_12. URL: http://dx.doi.org/10.1007/3-540-48256-3_12.

[22] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[23] Martin Davis, Georg Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[24] Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL.*

[25] Florian Haftmann and Tobias Nipkow. "Code Generation via Higher-Order Rewrite Systems". English. In: *Functional and Logic Programming.* Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 103–117. ISBN: 978-3-642-12250-7. DOI: 10.1007/978-3-642-12251-4_9. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_9.

[26] Filip Maric. "Formalization and Implementation of Modern SAT Solvers". In: *J. Autom. Reasoning* 43.1 (2009), pp. 81–119. DOI: 10.1007/s10817-009-9127-8.

[27] Filip Maric. "Formal Verification of Modern SAT Solvers". In: *Archive of Formal Proofs* (July 2008). http://afp.sf.net/entries/SATSolverVerification.shtml, Formal proof development. ISSN: 2150-914x.

[28] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. "Versat: A Verified Modern SAT Solver". In: *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation.* VMCAI'12. Philadelphia, PA: Springer-Verlag, 2012, pp. 363–378. ISBN: 978-3-642-27939-3.

[29] Tjark Weber. "SAT-based Finite Model Generation for Higher-Order Logic". PhD thesis. Germany: Institut für Informatik, Technische Universität München, Apr. 2008. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20081018-676608-1-8.