Contents

Draft
Nikolaj Bjørner, Leonardo de Moura.
To appear in a volume on *Advances in Tractability*
Editors: Lucas Bordeaux, Youssef Hamadi, Pushmeet Kohli,
Robert Mateescu.
May 29, 2012

1

Tractability and Modern Satisfiability Modulo Theories Solvers

1.1 Introduction

We discuss practice and theory of tractability of modern Satisfiability Modulo Theories, SMT, solvers. Our starting point is research and experiences in the context of the state-of-the art SMT solver Z3 [13], developed by the authors at Microsoft Research. We first cover a selection of the main challenges and techniques for making SMT solving practical, integrating algorithms for tractable sub-problems, and pragmatics and heuristics used in practice. We then take a proof-theoretical perspective on the power and scope of the engines used by SMT solvers. Most modern SMT solvers are built around a tight integration with efficient SAT solving. The framework is commonly referred to as DPLL(T), where T refers to a theory or a combination of theories. The theoretical result we present compares DPLL(T) with unrestricted resolution. A straight-forward adaption of DPLL(T) provides a weaker proof system than unrestricted resolution, and we investigate an extension we call *Conflict Directed Theory Resolution* as a candidate method for bridging this gap. Our results apply to the case where T is the theory of equality. Better search methods for other theories and their integration is a very active area of current research.

As a starting point, we will first briefly recall connections between SAT and SMT, connections between the solving methods used for SAT and SMT, and a short survey of some current applications of SMT solvers.

1.1.1 From SAT to SMT

SMT solving extends propositional satisfiability, a.k.a. SAT. The goal of SAT is to decide whether formulas over Boolean variables, formed using

logical connectives, are satisfiable by choosing a truth assignment for its variables. *LB: Let's keep in mind to add a reference to the planned chapter on SAT* SMT solvers allow a much richer vocabulary when creating formulas. Besides Boolean variables, formulas may contain general relations, such as equalities between terms formed from variables, free (uninterpreted) functions, and interpreted functions. The goal of SMT is to decide whether formulas over the richer SMT vocabulary are satisfiable by choosing an interpretation to the free variables and free functions of the formula. The *theories* provide the meaning for the interpreted functions. For example, the theory of arithmetic is commonly used in the context of SMT, and efficient solvers for arithmetic and other theories can be used to solve (arithmetical) constraints.

1.1.2 From DPLL to DPLL(T)

Modern SMT solvers have over the last decade relied on advances in modern SAT solvers. Not only is propositional satisfiability a special case of SMT, but a SAT solver based on the Davis-Putnam-Logeman-Logemann (DPLL) architecture [9] can be extended with theory solvers. Advances in SAT solvers include a better understanding of how to perform case splitting and learning useful lemmas during search using conflict directed clause learning. Section 1.5 recalls the DPLL(T) calculus and contains an abstract account for conflict directed clause learning. As we elaborate on below, one theoretical explanation for the efficiency of DPLL-based SAT solvers with conflict directed clause learning is the fact that it can simulate general resolution using at most a polynomial space overhead. On the other hand, DPLL search is more efficient: unit propagation corresponds to resolution, but unlike resolution, DPLL does not need to construct new clauses during propagation. There is so far no corresponding results in the context of SMT solvers. In fact solvers based on the architecture coined as DPLL(T) are exponentially worse off than resolution, even for the theory T of equality.

The connections between general resolution proofs and modern DPLL solvers have been the subject of several studies. It was shown in [28] that a DPLL calculus could be augmented with lemma learning to polynomially simulate unrestricted resolution [1]. In a quite similar context such clauses have been called *noogood*s in [42]. Modern DPLL-based SAT solvers are based on the conflict learning schemes introduced by the

[1] The terminology *unrestricted* resolution means that there are no ordering requirements or imposed strategies, such as the set of support strategy.

GRASP system [41]. GRASP uses a tight coupling of BCP (Boolean Constraint Propagation) by analyzing an implication graph for generating conflict clauses. The scheme is known as an *asserting clause learning scheme*. More recently it was shown that propositional DPLL with asserting clause learning schemes and restarts p-simulates general resolution proofs [36]. The notion of p-simulation comes from Karp reduction: If a proof in a formal system F_1 can be reduced to a proof in system F_2 using at most a polynomial space increase, then F_2 p-simulates F_1. We write $F_1 \equiv_p F_2$ if F_1 and F_2 can be reduced to each other. Their result strengthens several previous results [2, 7, 24] on connecting modern DPLL search with unrestricted resolution.

The state for SMT solvers is much less advanced. Progress in modern DPLL-based SAT solvers inspired developing efficient and scalable SMT solvers by plugging in theory solvers in a modular way. But it is well known that DPLL(T), and the realization of DPLL(T) in modern SMT solvers can be exponentially worse than general resolution systems, and various ad-hoc solutions, such as extending DPLL(T) [3] and exploring alternatives to DPLL(T) have been examined.

We will later develop an approach that we call *conflict directed theory resolution* to equip DPLL(T) with the ability to compete with general resolution systems. We examine the theory of equality in detail. This theory is fundamental in SMT solvers; formulas from several other theories can be reduced to formulas using only the theory of equality.

1.1.3 Applications

Thanks to technological advances and the ability to directly handle useful theories, SMT solvers are of growing relevance in the context of software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning and graph problems. Conversely, many features and optimizations in SMT solvers, such as Z3 [13], are based on the needs of applications. We cannot survey all applications here. We refer to [16] for more details. One important use is dynamic symbolic execution tools. They are used for generating inputs to unit tests and they can directly be used to expose bugs in security critical code. There are several tools, including CUTE, Exe/Klee, DART, Pex, SAGE, and Yogi [23]. These tools collect explored program paths as formulas and use SMT solvers to identify new test inputs that can steer execution into new branches. SMT solvers are a good fit for symbolic execution because the semantics of most program statements can be easily

modeled using theories supported by these solvers. The constraints generated from the tools are mostly conjunctions, but these can in principle still be as intractable as general formulas. Another important area is for static program analysis and verification tools. The ability to handle theories that are used in programming languages is also important here.

1.1.4 Outline

The rest of the chapter is structured into two parts. Sections 1.2-1.4 provide an introduction to tractability and SMT solving. The remainder provides a technical treatment of SMT solving for the theory of equality, which plays a central role in SMT.

Section 1.2 introduces SMT by an example. We then survey theories that are commonly used in SMT solvers in Section 1.3; some theories are tractable for conjunctions of atomic constraints, others are intractable, either NP hard or undecidable. Theoretical limits are not the only factors in SMT solvers. Section 1.4 discusses some practical features in SMT solvers that are useful for dealing with complexity.

Section 1.5 presents the main engine in modern SMT solvers as an abstract transition system. We also present a decision procedure for the theory of equality as an abstract inference system in Section 1.6. Alternatively, one can eliminate the theory of equality entirely as shown in Section 1.7; or adapt a hybrid scheme that applies to equalities under function applications (Section 1.8) and transitivity of equality (Section 1.9). Section 1.10 seeks a theoretical justification for the hybrid scheme: the resulting proof system corresponds closely to a proof system based on unrestricted resolution. We summarize the chapter in the conclusions 1.11.

1.2 SMT - an Appetizer

We will introduce three theories used in SMT solvers using the following example:

$$b + 2 \simeq c \ \wedge \ f(\mathit{read}(\mathit{write}(a, b, 3), c - 2)) \not\simeq f(c - b + 1).$$

The formula uses an un-interpreted function symbol f and the theories of arithmetic and arrays. The theory of arrays was introduced by McCarthy in [31] as part of forming a broader agenda for a calculus of computation. In the theory of arrays, there are two functions *read* and *write*. The

term $read(a, i)$ produces the value of array a at index i, while the term $write(a, i, v)$ produces an array, which is equal to a except for possibly index i which maps to v. These properties can be summarized using the axioms:

$$\forall i\, v\, a\, (read(write(a, i, v), i) \simeq v) \tag{1.1}$$

$$\forall i\, j\, v\, a\, (i \not\simeq j \;\rightarrow\; read(write(a, i, v), j) \simeq read(a, j)) \tag{1.2}$$

They state that the result of reading $write(a, i, v)$ at index j is v for $i \simeq j$. Reading the array at any other index produces the same value as $read(a, j)$.

On the other hand, the only thing we know about the un-interpreted function f is that for all t and s, if $t \simeq s$, then $f(t) \simeq f(s)$ (congruence rule). The congruence rule holds for both interpreted and un-interpreted functions and relations. It implies that formulas remain equivalent when replacing equal terms. The example formula is unsatisfiable. That is, there is no assignment to the integers b and c and the array a such that the first equality $b + 2 \simeq c$ holds and at the same time the second disequality also is satisfied. One way of establishing the unsatisfiability is by replacing c by $b + 2$ in the disequality, to obtain the equivalent

$$b + 2 \simeq c \;\wedge\; f(read(write(a, b, 3), b + 2 - 2)) \not\simeq f(b + 2 - b + 1),$$

which after reduction using facts about arithmetic becomes

$$b + 2 \simeq c \;\wedge\; f(read(write(a, b, 3), b)) \not\simeq f(3).$$

The theory of arrays implies that the nested array *read*/*write* functions reduce to 3 and the formula becomes:

$$b + 2 \simeq c \;\wedge\; f(3) \not\simeq f(3).$$

The congruence property of f entails that the disequality is false. As the example indicates, a main challenge in SMT solvers is to efficiently integrate a collection of theory solvers. The solvers cooperate checking satisfiability of formulas that can mix several theories. Formulas are also in general built from conjunctions, disjunctions, negations and other logical connectives, so the solvers are also required to work efficiently with logical formulas.

Most modern SMT solvers employ theory solvers that work with *conjunctions* of atomic constraints over the theory. So the theory solvers typically do not have to worry about propositional search. The propositional search is taken care of by state-of-the-art methods for propositional satisfiability. Section 1.5 contains a formal presentation of the most widely

used integration. Important capabilities of solvers in this context include handling incremental addition and deletion of constraints and to efficiently propagate truth assignments. We here mostly discuss solving quantifier-free formulas, but integrating reasoning for quantified formulas is also of high importance for applications of SMT solvers.

1.3 Tractable and Intractable Theories

As exemplified in the previous section, theories provide the meaning of a collection of functions and relations. Formally, a theory is defined by a *signature* that defines the domains, functions and relations of the theory and a set of interpretations of the relations and functions. We also suggested that theory solvers deal with conjunctions of atomic constraints over the theory. In the following let us examine some common theories and recall the complexity of solving conjunctions of atomic constraints.

1.3.1 Tractable Theories

We first recall the theories of equality and theory of linear real arithmetic that are tractable.

Example 1 [*UF* - Equality and Free Functions] *UF* is really best characterized as the *empty* theory. The signature comprises of functions and relations that are un-interpreted. In other words, the functions are not given any interpretation or constrained a priori in any other way. The only properties that hold are that equality $\simeq$ is an equivalence relation and each function f (henceforth for simplicity assumed binary) respects congruences. In other words, if f is applied to equal arguments, then the results are equal. These properties are common to all terms, whether they belong to a proper theory or not. They can be characterized as the set of inference rules:

$$\frac{v \simeq v', u \simeq u'}{f(u,v) \simeq f(u',v')} \qquad \frac{u \simeq v, v \simeq w}{u \simeq w} \qquad \frac{u \simeq v}{v \simeq u} \qquad \frac{}{v \simeq v} \tag{1.3}$$

Checking a conjunction of equalities and disequalities for satisfiability is tractable. The Downey-Tarjan-Sethi congruence closure algorithm [18] forms the basis of an $O(n \log n)$ algorithm for checking satisfiability, where n is the number of sub-terms in the conjunction.

□

A theory that is axiomatized using ground *Horn*-clauses is also tractable with respect to satisfiability of conjunctions of equalities and disequalities. An example Horn clause is $f(a,b) \simeq c \wedge f(b,a) \simeq d \rightarrow f(a,a) \simeq f(c,d)$. It has at most one positive equality and other equalities are negated. The polynomial saturation algorithm is obtained by using congruence closure as a sub-routine. Whenever the current set of equalities make all antecedents in a Horn-clause true, then the consequent equality is asserted. Section 1.7 presents an alternative construction, the Ackermann reduction, that shows Horn equalities to be tractable: it reduces Horn clauses with equality to propositional Horn clauses.

The theory of (Horn) equality is also known to be a *convex* theory [34]. Convex theories enjoy the property that if a conjunction of constraints imply a disjunction of equalities, then at least one of the equalities is already implied. LRA, described next, is also convex.

Example 2 [LRA - Linear arithmetic over the Reals] The signature of LRA is given by the domain R of reals, relations $\leq, <, \geq, >, \simeq$. The equality relation $\simeq$ is automatically contained in every theory. There is a constant r for every real number R. The operations are addition $+$, subtraction $-$, and multiplication by a constant r.

An example unsatisfiable conjunction of inequality constraints is $x \leq y + 1 \wedge y \leq z + 1 \wedge z < x - \frac{7}{3}$.

Satisfiability of conjunctions of inequalities can be reduced to linear programming feasibility. Thus, convex optimization techniques, such as Simplex and interior point methods, can be used to check for satisfiability of conjunctions of constraints.

Many modern SMT solvers use Dual Simplex [19] with support for efficient backtracking for LRA. A special case is when every constraint is of the form $x - y \leq r$: they contain two variables with unit coefficients 1 and -1. This case, a.k.a. *difference logic*, can be solved using efficient shortest path network algorithms, such as the Floyd-Warshall or the Ford Fulkerson algorithm [38, 39]. □

1.3.2 Intractable Theories

Many useful theories are intractable, even for conjunctions of atomic constraints. These include the theory of arrays, algebraic data-types, arithmetic involving integers and multiplication.

Example 3 [A - Arrays] The theory of arrays as formulated with formulas (1.1) and (1.2) is known as the *theory of non-extensional arrays*.

Checking conjunctions of equality and inequality constraints for satisfiability in this theory is NP hard. Take for instance a clause C_1: $(a \vee \neg b)$. We will use the array M to encode a propositional model, and for each clause C_i use a corresponding array to encode selecting a literal that evaluates to true in the model. In the context of the single clause, the formula looks as follows:

$$\begin{aligned} M &\simeq \mathit{write}(\mathit{write}(\mathit{write}(\mathit{write}(M', \ell_a, v_a), \ell_b, v_b), \ell_{\overline{a}}, v_{\overline{a}}), \ell_{\overline{b}}, v_{\overline{b}}) \\ v_a &\not\simeq v_{\overline{a}},\ v_b \not\simeq v_{\overline{b}},\ \mathtt{distinct}(\ell_a, \ell_b, \ell_{\overline{a}}, \ell_{\overline{b}}) \\ \mathit{read}(M, \mathit{sel}_1) &\simeq 1 \not\simeq \mathit{read}(M', \mathit{sel}_1) \\ C_1 &\simeq \mathit{write}(\mathit{write}(C_1', \ell_a, v_a), \ell_{\overline{b}}, v_{\overline{b}}) \\ \mathit{read}(C_1, \mathit{sel}_1) &\simeq 1 \not\simeq \mathit{read}(C_1', \mathit{sel}_1) \end{aligned}$$

We have used the shorthand $\mathtt{distinct}(t_1, \ldots, t_n)$ for $\bigwedge_{1 \le i < j \le n} t_i \neq t_j$. The last line of the encoding ensures that the index sel_1 selects a literal from the clause C_1. Note that the disequality $\mathit{read}(C_1, \mathit{sel}_1) \not\simeq \mathit{read}(C_1', \mathit{sel}_1)$ ensures that sel_1 has to be one of the literals used to update C_1'. The same literal is selected from M, and the value v_ℓ associated with that literal is 1. We can add more clauses to this encoding using a similar style as for C_1. The disequalities on the values v_ℓ associated with literals ensures that clauses are satisfied using the same assignment to the literals.

So what should a solver for the theory of arrays do about this? The approach we take in Z3 is to *reduce* the theory of arrays to the theory UF of uninterpreted functions [14]. The main idea of the reduction is to instantiate the array axioms (1.1) and (1.2) with instances that come from the formula being checked. There is a complete instantiation strategy for the theory of arrays that guarantees that a formula is satisfiable in the theory A of arrays if and only if the instantiation is satisfiable in UF. The size of the instantiated formula grows quadratically with the size of the original formula. Furthermore it introduces disjunctions, so unfortunately the efficient algorithms for checking conjunctions of atoms in UF cannot be applied alone. Heuristics are used to reduce the size of the resulting formula, and reduce the number of instantiations with disjunctions.

The non-extensional theory does not equate arrays that are equal with respect to *read*. The axiom of extensionality can be added and enforces these equalities:

$$\forall a\, b\, ((\forall \delta_{ab}\, \mathit{read}(a, \delta_{ab}) \simeq \mathit{read}(b, \delta_{ab})) \rightarrow a \simeq b) \tag{1.4}$$

The practical implications of adding extensionality incur a significant performance penalty in practice: in the limit, a satisfiable model has to determine whether each pair of array terms a, b can be distinguished using some index δ_{ab}. Good heuristics that avoid comparing arrays for equality when it is irrelevant, are therefore critical in this context. □

The terminology *reduction approach* was used by Kapur and Zarba in [26]. They survey several theories, including some mentioned here, whose decision problems can be reduced to *UF*.

Example 4 [CAL] *Combinatory Array Logic* [14] is an extension of the base theory extensional of arrays. Besides *write* and *read*, it admits functions *const* and for every function f there is a function map_f. The term $const(v)$ is an array that evaluates to v on every index, and $map_f(a, b)$ has the same arity as f (in this example f is binary), and maps f on the range of a, b. In other words, we have the axiomatic characterization:

$$\forall i\, v\, (read(const(v), i) \simeq v) \tag{1.5}$$

$$\forall i\, a\, b\, (read(map_f(a, b), i) \simeq f(read(a, i), read(b, i))) \tag{1.6}$$

CAL is also reducible to *UF* by instantiating the theory axioms. It is tempting to also add an identity array combinator "*id*" with the property $read(id, x) \simeq x$, satisfiability of the resulting decision problem for quantifier free formulas over this theory is highly intractable: it is undecidable. □

Example 5 [D - Algebraic Data-types] The quintessential algebraic data-type is the theory of pure LISP S-expressions. There are two constructors `cons` and `nil`. Everything that is a pure S-expression is either a `cons` or the constant `nil`, but not both. Terms that are `cons` are uniquely decomposed into a `car` (head) and a `cdr` (tail) portion. These are again S-expressions. Inductive data-types furthermore have to be *well-founded*. They correspond to finite trees. The first-order theory of (first-order) algebraic data-types was shown decidable by Malcev [29]. It comes with a non-elementary complexity: the complexity of the decision problem is a tower of exponentials, where the height of the tower is given by the number of alternations of quantifiers. Oppen [35] developed efficient algorithms for ground satisfiability for a theory of S-expressions when it can be assumed that `car(nil)` = `nil` = `cdr(nil)`. Without this assumption, the decision complexity of quantifier-free conjunctions of equalities and disequalities of data-type constraints is NP complete.

Tractability also gets lost when considering other kinds of data-types than S-expressions.

In Z3, the theory of algebraic data-types is, just like the theory of arrays, also be reduced to *UF* by adding enough instantiations of first-order axioms. To ensure well-foundedness the decision procedure performs unification-style occurs checks and adds axioms whenever the check fails. □

Example 6 [LIA - Linear integer arithmetic] Linear arithmetic over the domain of integers is also decidable, but this time satisfiability of conjunctions of inequalities is already well known to be NP complete [22]. □

Example 7 [NRA - Non-linear real arithmetic] Non-linear arithmetic over the reals admits multiplication between arbitrary terms, not just multiplication by constants. For example, the formulas $x \cdot x < 0$ is clearly unsatisfiable because every square is non-negative. NRA formulas with quantifiers are decidable using Collin's Cylindric Algebraic Decomposition algorithm [8] or Cohen-Hörmander's sign-based algorithm [27]. The theory is intractable, though still decidable. Designing practically efficient decision procedures, even for quantifier free formulas, is an ongoing challenge. □

Example 8 [NIA - Non-linear integer arithmetic] Non-linear arithmetic over the integers is already undecidable for Diophantine equations (Hilbert's tenth problem) [30], and Gödel's celebrated incompleteness result for arithmetic establishes that there is no recursive axiomatization when adding quantifiers. □

There are many other theories of relevance for SMT solvers. These include theories of Boolean algebras (sets), multi-sets, strings and sequences, queues (sequences where you can add and remove elements from both ends), regular languages, bit-vectors and lists and data-structures with reachability predicates. The theoretical complexities for conjunctions range from linear to highly intractable. The theory *UF* of uninterpreted functions is a base theory that many other theories, such as A, CAL and D, can be reduced to.

1.4 Practice and Pragmatics

Theoretical tractability matters, but other factors have an even more significant influence on what makes SMT solving feasible for problems that come from applications. We discuss some here.

1.4.1 Simplification

In many situations, SMT solvers are supplied with formulas that are generated by a tool. The formulas often contain assertions that have nothing to do with the main property. In other cases, assertions are equalities that can be solved, and the solution allows to eliminate variables from the search space. An important component of Z3 therefore comprises of pre-processing simplification routines that reduce the assertions.

1.4.2 Polynomial factors matter

Consider the two numbers: 2^{10} and 2^{10^2}. A problem instance of size 10 is solvable even if it is handled by a procedure that takes 2^{10} milliseconds (a second), but it is not solvable for a procedure that takes 2^{10^2} milliseconds (40,196,936,841,331,475,186 years). Of course, even algorithms that are quadratic time are impractical when applied to inputs of modest size. For example, using insertion sort on an array with 100,000 elements is impractical. When combining two separate theory solvers for disjoint signatures, SMT solvers need to exchange equalities between the two solvers. With potentially one equality for every pair of terms there is a quadratic worst case number of equalities to share. Model-based theory combination [12] uses the fact that most theory solvers build and maintain partial models. Only equalities between terms that are equal in the current partial model are shared. The worst case is still quadratic, but the average case behavior we have observed on applications is far better.

1.4.3 Relevancy Propagation

Z3, similar to the Simplify theorem prover [17] and several other SMT solvers, uses quantifier instantiation to convert formulas with quantifiers into quantifier-free formulas that are handled by ground decision procedures [10]. It uses terms from the quantifier-free assertions to produce new quantifier instances. Each new term may lead to additional instantiations. So terms from sub-formulas that do not contribute to

satisfiability can lead to an inhibiting large search space. Relevancy propagation [11] keeps track of which truth assignments are essential for determining satisfiability of a formula. Atoms that are marked as *relevant* have their truth assignment propagated, while atoms that are not marked as relevant do not participate in propagation and do not contribute to quantifier instantiation. We found relevancy filtering useful for quantified formulas, but there is a trade-off: for quantifier-free formulas, it hides potentially useful lemmas.

1.4.4 Reducing Proof and Model Search

Quantified formulas often bind more than one variable. Even if we restrict the set of possible instantiations per quantified variable to a smaller finite set, the number of possible instantiations of a quantifier grows exponentially in the number of different quantified variables. The main method used in SMT solvers for throttling quantifier instantiations is by using *patterns* that control which instantiations are used. The patterns can be used to enforce dependencies between instantiated variables.

A different mechanism with a related aim is to restrict the space of interpretations of relations by using *templates* [15]. The idea is to replace uninterpreted functions and relations by specializations that constrains the set of possible interpretations. For example, we can replace the relation $R(x, y, z)$ by the relation $(R_1(x, y) \wedge y \simeq z) \vee (R_2(x, z) \wedge x \simeq y)$. The resulting formula could become unsatisfiable, but a search for satisfiable instances is on the other hand simpler.

1.5 DPLL(T) - a Framework for Efficient SMT Solvers

As preparation for the theoretical treatment we will now describe the algorithmic underpinnings of modern SMT solvers. Most modern SMT solvers are built around the DPLL(T) architecture that provides a tight integration with efficient SAT solving. Figure 1.1 shows an abstract reconstruction of DPLL(T).

It follows [21], with one difference: we include explicit transitions for conflict resolution. The DPLL(T) procedure is modeled as an abstract transition system. There are two kinds of states. The *search* states are of the form $M \parallel F$, where M is a partial assignment given as a stack of literals, and F is a set of set of clauses. A clause is often referred to as

Initialize	$\Longrightarrow \; \parallel F$	F is a set of clauses
Decide	$M \parallel F \;\Longrightarrow\; M\ell^d \parallel F$	**if** $\begin{cases} \ell \text{ or } \neg\ell \text{ occurs in } F \\ \ell \text{ unassigned in } M \end{cases}$
Propagate	$M \parallel F, C \vee \ell \;\Longrightarrow\; M\ell^{C\vee\ell} \parallel F, C \vee \ell$	**if** $\begin{cases} \ell \text{ unassigned in } M \\ \overline{C} \subseteq M \end{cases}$
Conflict	$M \parallel F, C \;\Longrightarrow\; M \parallel F, C \parallel C$	**if** C is false under M
Resolve	$M \parallel F \parallel C' \vee \neg\ell \;\Longrightarrow\; M \parallel F \parallel C \vee C'$	**if** $\ell^{C\vee\ell} \in M$
Backjump	$M\ell_0^d M' \parallel F \parallel C \vee \ell \;\Longrightarrow\; M\ell^{C\vee\ell} \parallel F, C \vee \ell$	**if** $\overline{C} \subseteq M, \neg\ell \subseteq \ell_0^d M'$
Restart	$M \parallel F \;\Longrightarrow\; \parallel F$	
Unsat	$M \parallel F \parallel \emptyset \;\Longrightarrow\; \texttt{unsat}$	
Sat	$M \parallel F \;\Longrightarrow\; M$	**if** $\begin{cases} M \text{ is } T \text{ consistent} \\ \text{and is a model for } F. \end{cases}$
T-Propagate	$M \parallel F \;\Longrightarrow\; M\ell^{C\vee\ell} \parallel F$	**if** $\begin{cases} \ell \text{ unassigned in } M \\ \ell \text{ or } \neg\ell \text{ occurs in } F \\ T \vdash C \vee \ell \\ \overline{C} \subseteq M \end{cases}$
T-Conflict	$M \parallel F \;\Longrightarrow\; M \parallel F \parallel C$	**if** $\overline{C} \subseteq M$, $T \models C$.

Figure 1.1 Abstract DPLL(T) Procedure

C. It is a disjunction of literals that we refer to as ℓ (which is either an atom p or a negation of an atom $\neg p$). The set $\overline{C}$ is obtained from a clause C by negating all the literals in C. The *conflict resolution* states are of the form $M \parallel F \parallel C$, where C is a conflict clause. We will later make use these two kinds of states when formulating new rules.

Search starts with the Initialize rule that creates a state $\parallel F$. Search proceeds by a sequence of decision and propagation steps until either reaching a conflict or a satisfiable assignment. Conflicts trigger the conflict resolution rules to be applied. We use two kinds of annotations for the literals in M. Literals annotated as ℓ^d are *decision literals*. They

are assigned by the Decide rule. Literals can also be annotated with a clause, so they are of the form $\ell^{(C \vee \ell)}$. The literal ℓ occurs positively in the clause. These literals are added to M as a result of Propagate (or T-Propagate). The clause is later used for conflict resolution. There are two rules that distinguish DPLL(T) from modern DPLL. These are the T-Propagate and T-Conflict rules. The rules describe the main ways for theories to integrate with DPLL. The T-Propagate rule lets theories participate in unit propagation and T-Conflict lets theories determine when a clause is conflicting under the partial assignment M and the theory T. Note that the rules maintain the set of literals from the input F. It is therefore easy to establish that DPLL(T) is terminating (assuming Restart is applied using a back-off).

The condition on Backjump is crucial. It captures how modern DPLL solvers filter which clauses to learn, it is called an *asserting scheme*, and at the same time provides a scheme for backtracking. It states that there is precisely one literal ℓ in the clause $C \ \vee \ \ell$, forced false by the last decision literal ℓ_0^d. Some variations of these rules are possible. For example, one can decouple clause learning from back-jumping, and admit garbage collection of learned clauses.

Example 9 (A Derivation) *To give a feel for DPLL(T), let us prove the following theorem:*

$$\underbrace{f^2a \simeq a \wedge f^3b \simeq b \wedge a \not\simeq f(a)}_{M_0} \wedge \underbrace{(a \simeq b \ \vee \ a \simeq c) \wedge (a \not\simeq c \ \vee \ f^4a \simeq b)}_{F}$$

where we used the abbreviation f^2a for $f(f(a))$. We have indicated a state $M_0 \parallel F$ that is obtained by including the unit-literals in the partial model M_0. We continue a derivation by

$$\begin{array}{ll} & M_0 \parallel F \\ \Longrightarrow & \{\textsf{Decide } \textit{on } a \simeq b\} \\ & M_0(a \simeq b)^d \parallel F \\ \Longrightarrow & \{T\text{-Conflict}\} \\ & M_0(a \simeq b)^d \parallel F \parallel \underbrace{f^2a \simeq a \wedge f^3b \simeq b \wedge a \simeq b \rightarrow a \simeq f(a)}_{C_1} \\ \Longrightarrow & \{\textsf{Backjump}\} \\ & M_0(a \not\simeq b)^{C_1} \parallel F, C_1 \\ \Longrightarrow & \{\textsf{Propagate}\} \\ & M_0, (a \not\simeq b)^{C_1}, (a \simeq c)^{(a \simeq b \vee a \simeq c)} \parallel F, C_1 \end{array}$$

$\Longrightarrow$ {Propagate}

$\underbrace{M, (a \not\simeq b)^{C_1}, (a \simeq c)^{(a\simeq b \vee a \simeq c)}, (f^4 a \simeq b)^{(a \not\simeq c \vee f^4 a \simeq b)}}_{M_1} \parallel F, C_1$

$\Longrightarrow$ {T-Conflict}

$M_1 \parallel F, C_1 \parallel f^2 a \simeq a \wedge f^3 b \simeq b \wedge f^4 a \simeq a \rightarrow a \simeq f(a)$

$\Longrightarrow$ {Resolve *with* $(f^4 a \simeq b)^{(a \not\simeq c \vee f^4 a \simeq b)}$}

$M_1 \parallel F, C_1 \parallel f^2 a \simeq a \wedge f^3 b \simeq b \wedge a \simeq c \rightarrow a \simeq f(a)$

$\Longrightarrow$ {Resolve *with* $(a \simeq c)^{(a \simeq b \vee a \simeq c)}$}

$M_1 \parallel F, C_1 \parallel f^2 a \simeq a \wedge f^3 b \simeq b \rightarrow a \simeq b \vee a \simeq f(a)$

$\Longrightarrow$ {Resolve *with* $(a \not\simeq b)^{C_1}$}

$M_1 \parallel F, C_1 \parallel f^2 a \simeq a \wedge f^3 b \simeq b \rightarrow a \simeq f(a)$

$\Longrightarrow$ {Resolve *with* M_0}

$M_1 \parallel F, C_1 \parallel \emptyset$

$\Longrightarrow$ Unsat

□

1.6 Abstract Congruence Closure

We introduced the theory *UF* of equality in Example 1. Let us here present an abstract decision procedure for it.

Efficient congruence closure algorithms [18, 17, 33] compute the set of all implied equalities from a basis of asserted equalities by maintaining the coarsest equivalence relation $\sim$ that is closed under asserted equalities and the congruence rule. The rule Assert in Figure 1.2 encodes an asserted equality as a state that contains a node definition $r \equiv (u \simeq v)$ and asserted literal r. The congruence rule Cong is triggered whenever there are two nodes labeled by the same function f or relation and whose children are equivalent. Only parents of nodes whose equivalence class are updated need to be processed. Therefore, efficient implementations of this rule index each node by the sets of *parent* nodes where they occur. The other rules Trans and Sym are triggered implicitly by maintaining $\sim$ in a union-find data-structure.

To simplify notation we used a binary function f to represent arbitrary n-ary functions. It is of course possible to literally replace n-ary functions

$$\textsf{Assert}\ \frac{r \equiv (u \simeq v);\quad r}{u \sim v} \qquad \textsf{Cong}\ \frac{\begin{array}{c} w \equiv f(u,v),\ w' \equiv f(u',v'); \\ v \sim v', u \sim u' \end{array}}{w \sim w'}$$

$$\textsf{Trans}\ \frac{u \sim v,\ v \sim w}{u \sim w} \qquad \textsf{Sym}\ \frac{u \sim v}{v \sim u}$$

Figure 1.2 Abstract Congruence Closure

by $n-1$ binary functions and work entirely with binary functions, but our results do not use any special assumptions about binary functions.

1.7 The Ackermann Reduction

The Ackermann reduction lets us reduce the theory of equality to the theory of purely propositional logic. We describe a basic Ackermann reduction scheme here. The Ackermann reduction can also be seen as a basis of the optimizations we pursue in later sections.

For a fixed set of terms and their sub-terms, $u_1, u_2, \ldots, u_N$ there is a finite number of ways one can apply the congruence closure rules to derived implied equalities. It is therefore straight-forward to compile the congruence closure rules into a set of clauses using $\mathcal{O}(N^2)$ auxiliary equality predicates. The procedure ackermannize eliminates the function symbol f from a set of formulas F:

ackermannize(f, F) :
  foreach $f(u,v) \in F$ **where** u, v do not contain f
    create a fresh constant $a_{f(u,v)}$
    replace $f(u,v)$ by $a_{f(u,v)}$ in F
  foreach $a_{f(u,v)}, a_{f(u',v')}$
    add the clause $u \simeq u' \wedge v \simeq v' \ \rightarrow\ a_{f(u,v)} \simeq a_{f(u',v')}$ to F

Compilation allows reducing the theory *UF* for quantifier-free formulas to the theory of pure equalities or all the way to propositional SAT. Many optimizations to the basic Ackermann reduction have been proposed and used over the years, including, [5, 6, 32, 37, 4].

Let us just notice a trivial optimization here: we can avoid the symmetry rule by normalizing all equalities to the form $u_i \simeq u_j$ where $1 \leq i < j \leq N$. The clauses added by the Ackermann reduction are then

of the form:

$$u \simeq u' \ \wedge \ v \simeq v' \ \rightarrow \ a_{f(u,v)} \simeq a_{f(u',v')} \ \ \mathsf{Cong} \tag{1.7}$$

$$u \simeq v \ \wedge \ v \simeq w \ \rightarrow \ u \simeq w \ \ \mathsf{Trans} \tag{1.8}$$

Ackermannization has the disadvantage as the number of additional literals is in the worst case quadratic in the size of the input. It is furthermore a problem to eliminate function symbols using Ackermannization when the same function symbols could be re-introduced to the search space when quantifiers are instantiated incrementally during search.

1.8 Dynamic Ackermann Reduction

Section 1.6 presented an abstract account of a congruence closure based decision procedure for equality. It can be plugged into the DPLL(T) framework as a theory solver. Section 1.7 took a different perspective on the theory of equality; it presented a reduction to pure equalities (without function symbols) or propositional SAT.

Nevertheless, Ackermann reduction has the advantage of admitting exponentially shorter refutations than a DPLL(T) integration of congruence closure. The following formula has a short DPLL refutation after an Ackermann reduction, but does not have a short proof in DPLL(T).

$$\bigwedge_{i=1}^{N} (p_i \vee x_i \simeq v_0) \ \wedge \ (\neg p_i \vee x_i \simeq v_1) \ \wedge \ (p_i \vee y_i \simeq v_0) \ \wedge \ (\neg p_i \vee y_i \simeq v_1)$$
$$\wedge \ f(x_N, \ldots, f(x_2, x_1)\ldots) \not\simeq f(y_N, \ldots, f(y_2, y_1)\ldots) \tag{1.9}$$

In [20], an approach, called *Dynamic Ackermannization*, is proposed to cope with this problem. There, clauses corresponding to Ackermann's reduction are added when a congruence rule participates in a conflict. We can formulate the Dynamic Ackermann reduction as an inference rule that gets applied during conflict resolution.

$$\mathsf{Dyn.\ Cong} \quad \frac{M \parallel F \parallel C}{M \parallel F, (u \simeq u' \ \wedge \ v \simeq v' \ \rightarrow \ f(u,v) \simeq f(u',v')) \parallel C}$$

subject to a suitable filter that throttles its use.

(*) **if** the congruence rule is applied - repeatedly - to the premises $u \simeq u' \ \wedge \ v \simeq v'$ and conclusion $f(u,v) \simeq f(u',v')$, such that $f(u,v), f(u',v')$ occur in C.

This filter ensures that no new terms are introduced, but it may still introduce new equalities for either $u \simeq u'$, $v \simeq v'$ or $f(u,v) \simeq f(u',v')$. We will in the following use a stronger filter and call the rule associated with the stronger filter Dyn. Cong$^\sharp$. The stronger filter requires:

$$\begin{array}{ll} \textsf{Dyn. Cong}^\sharp & M \parallel F \parallel C \Longrightarrow \\ & \quad M \parallel F, (u \simeq u' \ \wedge \ v \simeq v' \ \rightarrow \ f(u,v) \simeq f(u',v')) \parallel C \\ & \quad \textbf{if } (*) \text{ and } \ f(u,v) \simeq f(u',v') \in C \end{array}$$

Dynamic Ackermannization allows DPLL(T) solvers to find short proofs for formulas, such as (1.9), but are not sufficient for DPLL(T) to find short proofs in cases where full Ackermann reduction applies. In the following, we describe a formula that remains hard, even in the context Dynamic Ackermann reduction.

1.9 The Price of Equality

Dynamic Ackermann reduction has advantages when used on formulas such as (1.9), but there are formulas with equality where DPLL(UF) still suffers from being incapable of generating short proofs. Consider the unsatisfiable formula (1.10) (and illustrated in Figure 1.3) also used in [37].

$$a_1 \not\simeq a_{50} \ \wedge \ \bigwedge_{i=1}^{49} [(a_i \simeq b_i \wedge b_i \simeq a_{i+1}) \ \vee \ (a_i \simeq c_i \wedge c_i \simeq a_{i+1})] \quad (1.10)$$

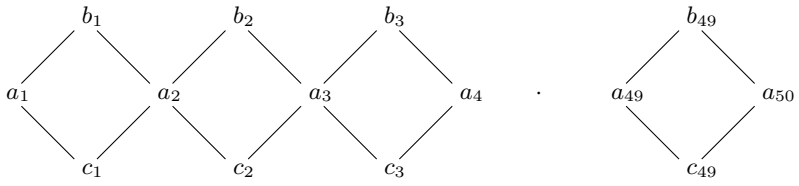

Figure 1.3 Diamond equalities

The formula is unsatisfiable because in every diamond, it is the case that $a_i \simeq a_{i+1}$ because either $a_i \simeq b_i \wedge b_i \simeq a_{i+1}$ or $a_i \simeq c_i \wedge c_i \simeq a_{i+1}$. Therefore, by repeating this argument for every i, we end up with the implied equality $a_1 \simeq a_{50}$. This contradicts the disequality $a_1 \not\simeq a_{50}$. A proof search method directly based on DPLL(UF) is not able to produce a succinct proof like the informal justification just given. Each of the

equalities $a_i \simeq b_i$, $b_i \simeq a_{i+1}$, $a_i \simeq c_i$, $c_i \simeq a_{i+1}$ and $a_1 \simeq a_{50}$ is *treated as an atom*. The atoms $a_i \simeq a_{i+1}$ are not present and DPLL assigns truth values only to the existing atoms. So a decision procedure for equalities detects a contradiction only when for every $i = 1, \ldots, 49$ $a_i \simeq a_{i+1}$ follows from either $a_i \simeq b_i \wedge b_i \simeq a_{i+1}$ or $a_i \simeq c_i \wedge c_i \simeq a_{i+1}$. There are 2^{49} different such equality conflicts, none of which subsumes the other. There is no short unsatisfiability proof that uses only the atoms in the original formula.

Yices and Z3 [3] include a method that uses the Join rule to propagate equalities.

$$\frac{\begin{array}{c} M p^d \,\|\, F \Longrightarrow M_1 \,\|\, F \qquad M \neg p^d \,\|\, F \Longrightarrow M_2 \,\|\, F \\ p \text{ is the only decision variable in } M_1, M_2 \end{array}}{M \,\|\, F \Longrightarrow M_1 \sqcup M_2 \,\|\, F}\ \textsf{Join}$$

The rule Join allows for splitting on a single atom p. The implied consequences of the different cases for p are then combined. We say that this approach uses *one look-ahead*. It bears similarities to the dilemma rule [40] known from SAT. One look-ahead is not always sufficient for learning the right implied facts. The join rule is also applied at the *base level*, that is, when M does not contain any decision variables. The new atoms are then not used in case splits. Consider a simple extension of the diamond problem given in equation (1.11), and illustrated in Figure 1.4.

$$a_1 \not\simeq a_{50} \ \wedge \ \bigwedge_{i=1}^{49} \left[\begin{array}{ll} & (a_i \simeq b_i \wedge b_i \simeq a_{i+1}) \\ \vee & (a_i \simeq c_i \wedge c_i \simeq a_{i+1}) \\ \vee & (a_i \simeq d_i \wedge d_i \simeq a_{i+1}) \end{array} \right] \tag{1.11}$$

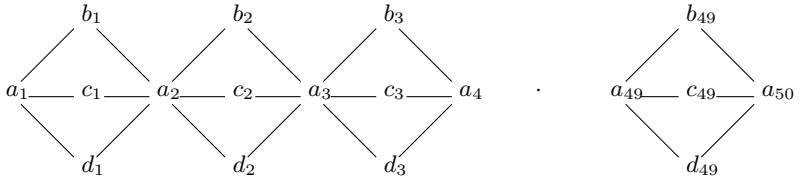

Figure 1.4 Double Diamond equalities

The Join rule is ineffective at finding short proofs for this and many other cases.

Following the style of dynamic Ackermann reduction we can formulate a rule that does introduce useful lemmas and literals for such cases. We call the rule Dyn. Trans:

$$\textsf{Sup}\ \frac{C \vee a \simeq b \qquad D \vee \ell[a]}{C \vee D \vee \ell[b]} \qquad \textsf{Res}\ \frac{C \vee \ell \qquad D \vee \overline{\ell}}{C \vee D}$$

$$\textsf{E-Dis}\ \frac{C \vee a \not\simeq a}{C} \qquad \textsf{Factor}\ \frac{C \vee \ell \vee \ell}{C \vee \ell} \qquad \textsf{E-Eqs}\ \frac{C \vee a \simeq b \vee a \simeq c}{C \vee a \simeq b \vee b \not\simeq c}$$

Figure 1.5 E-Res: Ground E-resolution calculus

$$\textsf{Dyn. Trans} \quad M \parallel F \parallel C \Longrightarrow M \parallel F, (u \simeq v \wedge v \simeq w \rightarrow u \simeq w) \parallel C \qquad u \not\simeq v, v \not\simeq w \in C$$

1.10 Conflict Directed Equality Resolution

We have identified two problems with DPLL(UF) where it is unable to find short proofs, and we have presented two rules Dyn. Trans and Dyn. Cong$^\sharp$ that overcome the problems. They introduce new literals, but use a throttle based on conflict resolution to limit the set of new literals. How effective are our additions to DPLL(UF)?

To characterize it, we will recall an unrestricted calculus for equality resolution, called E-Res. The purpose of this section is to establish that

$$\textsf{E-Res} \equiv_p \text{DPLL}(UF) + \textsf{Dyn. Cong}^\sharp + \textsf{Dyn. Trans.}$$

In other words, DPLL(UF) augmented with the two rules for dynamically adding clauses p-simulates resolution style proof systems for quantifier-free equality.

Figure 1.5 shows a ground E-resolution calculus, E-Res. In contrast to super-position calculi for non-ground clauses there are no ordering filters to reduce the set of rule applications. The rules can be applied without ordering filters. This makes the proof system more liberal than ordered super-position calculi, the search space is much bigger, but it admits proofs where the ordered version requires exponential more space [3].

1.10.1 Analysis

Our plan is to simulate E-Res. We first establish that every literal used in an E-Res proof can be produced using a polynomial overhead using the calculus obtained from using congruence closure as a theory solver + Dyn. Cong$^\sharp$ + Dyn. Trans. We call this calculus CDER (for conflict

directed equality resolution). Our results will be established relative to an assumption that is required for our analysis:

Assumption 1 *Every derivation in* E-Res *using only unit clauses can be produced as a T-Conflict by CDER.*

In other words, whenever there is an E-Res proof of a conflict using a set of equalities and one disequality, then that same set, and not a subset, is produced by CDER as a conflict clause. Second, we establish that every E-Res proof can be converted into a propositional resolution proof with at most a polynomial overhead. Third, we apply already established results on how DPLL+CL+Restarts p-simulate unrestricted resolution. The relevant propositions are summarized as follows:

Proposition 1 *Given an* E-Res *proof Π, then every literal that occurs in Π can be produced from either the input clauses Δ or using a polynomial number of applications of CDER deriving clauses Δ' that imply the additional literals.*

Proposition 2 *Given an* E-Res *proof Π, there is a proof Π' of size polynomial in Π whose support is $\Delta \cup \Delta'$ that uses only resolution.*

Proposition 3 ([36]) *For any asserting scheme, DPLL+CL+Restarts p-simulates unrestricted resolution.*

Let us note that these results rely on heuristics for controlling variable splitting, restarts and when to apply the resolution rules for guiding the proof search. Searching for proofs is still hard in worst case [1].

We will now establish the first two propositions. Let Π be an E-Res proof of the empty clause, then we will establish that every literal that occurs in a clause Π is eventually removed. Let Π' be a sub-tree of Π that introduces the literal ℓ (there could be more than one, but we can consider one at a time). We will first establish that for an inference rule that introduces a fresh literal in Π there is a sequence of theory conflict resolution steps that (1) produce the new literal, (2) allow replacing the inference rule by propositional resolution. In this context consider the rules Res and E-Dis as the ones that are responsible for removing literals. Given an inference with sub-tree Π' we will define a *core* set of literals that can be used to produce the literal introduced by Π'.

Definition 1 (Literal core) *Given a sub-tree Π', define:*

$$core\,(\pi, \ell, \Pi)_{\Pi'} = \emptyset \quad \text{if } \Pi = \Pi'$$

Otherwise,

$$core\left(\pi, \ell, \ \frac{C \in \Delta}{C}\ \right)_{\Pi'} = \{\pi\ell\}$$

$$core\left(\pi, \ell[b], \ \frac{\dfrac{\Pi_1}{C \ \vee \ a \simeq b} \qquad \dfrac{\Pi_2}{D \ \vee \ \ell[a]}}{C \ \vee \ D \ \vee \ \ell[b]}\ \right)_{\Pi'} =$$

$$core\left(+, a \simeq b, \ \frac{\Pi_1}{C \ \vee \ a \simeq b}\ \right)_{\Pi'} \cup \quad core\left(\pi, \ell[a], \ \frac{\Pi_2}{D \ \vee \ \ell[a]}\ \right)_{\Pi'}$$

$$core\left(+, b \not\simeq c, \ \frac{\dfrac{\Pi}{C \ \vee \ a \simeq b \ \vee \ a \simeq c}}{C \ \vee \ a \simeq b \ \vee \ b \not\simeq c}\ \right)_{\Pi'} =$$

$$core\left(+, a \simeq b, \ \frac{\Pi}{C \ \vee \ a \simeq b \ \vee \ a \simeq c}\ \right)_{\Pi'} \cup$$
$$core\left(-, a \simeq c, \ \frac{\Pi}{C \ \vee \ a \simeq b \ \vee \ a \simeq c}\ \right)_{\Pi'}$$

$$core\left(\pi, \ell, \ \frac{\Pi_1 \qquad \Pi_2}{\ell \ \vee \ C}\ \right)_{\Pi'} = \begin{array}{ll} & (core\,(\pi, \ell, \Pi_1)_{\Pi'} \ \mid \ell \in \Pi_1) \\ \cup & (core\,(\pi, \ell, \Pi_2)_{\Pi'} \ \mid \ell \in \Pi_2) \end{array}$$

$$core\left(\pi, \ell, \ \frac{\dfrac{\Pi}{\ell \ \vee \ D}}{\ell \ \vee \ C}\ \right)_{\Pi'} = core\left(\pi, \ell, \ \frac{\Pi}{\ell \ \vee \ D}\ \right)_{\Pi'}$$

Finally, let:

$$trail\left(\ \frac{\dfrac{\Pi_1}{C \vee \ell} \qquad \dfrac{\Pi_2}{D \vee \overline{\ell}}}{C \vee D}\ \right)_{\Pi'} =$$
$$core\left(+, \ell, \frac{\Pi_1}{C \vee \ell}\right)_{\Pi'} \cup core\left(+, \overline{\ell}, \frac{\Pi_2}{D \vee \overline{\ell}}\right)_{\Pi'}$$

$$trail\left(\ \frac{\dfrac{\Pi}{C \vee a \not\simeq a}}{C}\ \right)_{\Pi'} = core\left(+, a \not\simeq a, \ \frac{\Pi}{C \vee a \not\simeq a}\ \right)_{\Pi'}$$

Note that the rules for the ground resolution calculus introduce literals. The rule Sup introduces the literal $\ell[b]$ and rule E-Eqs introduces the literal $b \not\simeq c$. A derivation Π' that ends with Sup introduces the literal $\ell[b]$. A derivation Π' that ends with E-Eqs introduces the literal $b \not\simeq c$.

Lemma 1 *Let ℓ be the literal introduced by the rule Π' and let Π be the corresponding descendant of Π' that eliminates ℓ. The computation of $trail\,(\Pi)_{\Pi'}$ consists of recursive calls of the form $core\,(\pi, \ell', \Pi'')_{\Pi'}$. For any such recursive call:*

$$\bigwedge core\,(\pi, \ell', \Pi'')_{\Pi'} \wedge \ell \rightarrow \ \pi\ell'$$

Proof: The proof proceeds by induction over the calls to *core* with measure $|\Pi|$. Let us illustrate the base cases and one case of induction. The first base case is $\bigwedge core\,(\pi, \ell, \Pi')_{\Pi'} \wedge \ell \;\rightarrow\; \ell$, since $\bigwedge core\,(\pi, \ell, \Pi')_{\Pi'} = \bigwedge \emptyset = \mathit{true}$. The second base case is when $core\left(\pi, \ell', \;\dfrac{C \in \Delta}{C}\;\right)_{\Pi'} = \{\pi\ell'\}$. In this case the implication we need to establish is $\pi\ell' \wedge \ell \rightarrow \pi\ell'$. Now consider E-Eqs as an example,

$$\bigwedge core\left(+, a \simeq b, \;\frac{\Pi}{C \;\vee\; a \simeq b \;\vee\; a \simeq c}\;\right)_{\Pi'} \wedge \ell \rightarrow a \simeq b$$

$$\bigwedge core\left(-, a \simeq c, \;\frac{\Pi}{C \;\vee\; a \simeq b \;\vee\; a \simeq c}\;\right)_{\Pi'} \wedge \ell \rightarrow a \not\simeq c$$

The conjunction of the two premises imply that $b \not\simeq c$, which is what the lemma requires. □

Lemma 2 *For every literal ℓ introduced by the rule Π' there is a descendant Π such that*

$$\ell, \mathit{trail}\,(\Pi)_{\Pi'} \vdash \mathit{false}$$

Furthermore, if $\ell = \ell[b],\; \Pi' = \dfrac{\dfrac{\Pi_1}{C \;\vee\; a \simeq b} \quad \dfrac{\Pi_2}{D \;\vee\; \ell[a]}}{C \;\vee\; D \;\vee\; \ell[b]}$ *, then*

$$a \simeq b, \ell[a], \mathit{trail}\,(\Pi)_{\Pi'} \vdash \mathit{false}$$

and if $\ell = b \not\simeq c,\; \Pi' = \dfrac{\dfrac{\Pi_1}{C \;\vee\; a \simeq b \;\vee\; a \simeq c}}{C \;\vee\; a \simeq b \;\vee\; b \not\simeq c}$ *, then*

$$a \simeq b,\; a \not\simeq c, \mathit{trail}\,(\Pi)_{\Pi'} \vdash \mathit{false}$$

Proof: We first establish that $\ell \wedge \bigwedge \mathit{trail}\,(\Pi)_{\Pi'} \rightarrow \mathit{false}$, by unfolding the definition of *trail*. Lemma 1 implies that

$$\ell \wedge \bigwedge core\left(+, \ell', \frac{\Pi_1}{C \vee \ell'}\right)_{\Pi'} \wedge \bigwedge core\left(+, \overline{\ell'}, \frac{\Pi_2}{D \vee \overline{\ell'}}\right)_{\Pi'} \;\rightarrow\; \ell' \wedge \neg\ell'$$

and that

$$\ell \wedge \bigwedge core\left(+, a \not\simeq a, \;\frac{\Pi}{C \vee a \not\simeq a}\;\right)_{\Pi'} \;\rightarrow\; a \not\simeq a$$

The antecedents are contradictory in both cases. The two other claims in the lemma follow because $\ell[b]$ is implied by $a \simeq b, \ell[b]$ and $b \not\simeq c$ is implied by $a \simeq b, a \not\simeq c$. □

Lemma 2 implies that literals that are introduced by the rules Sup and E-Eqs participate in an E-conflict. Proposition 1 is a consequence of lemmas 1 and 2. We can now prove the next proposition.

Proof: [Of Proposition 2] Let us consider the inference rules that introduce new literals and rewrite the inferences to propositional resolution modulo lemmas produced from theory resolution.

Sup rewrites one side of an equality (the literal $\ell[a]$ is of the form $a \simeq c$ for some term c):

$$\frac{C \ \vee \ a \simeq b \qquad D \ \vee \ a \simeq c}{C \ \vee \ \ D \ \vee \ b \simeq c} \ \mapsto$$

$$\frac{C \ \vee \ a \simeq b \qquad \dfrac{a \simeq b \ \wedge \ a \simeq c \ \rightarrow b \simeq c \qquad D \ \vee \ a \simeq c}{D \ \vee \ a \not\simeq b \ \vee \ b \simeq c}}{C \ \vee \ D \ \vee \ b \simeq c}$$

The required lemma $a \simeq b \ \wedge \ a \simeq c \ \rightarrow b \simeq c$ is produced by theory resolution from the conflict that includes the two premises $a \simeq b$ and $a \simeq c$.

Sup rewrites a nested occurrence of a in a term that occurs in an equality or disequality:

$$\frac{C \ \vee \ a \simeq b \qquad D \ \vee \ t[a] \simeq s}{C \ \vee \ \ D \ \vee \ t[b] \simeq s} \ \mapsto$$

$$\frac{C \ \vee \ a \simeq b \qquad \dfrac{a \simeq b \ \rightarrow t[a] \simeq t[b] \qquad D \ \vee \ t[a] \simeq s}{D \ \vee \ a \not\simeq b \ \vee \ t[b] \simeq s}}{C \ \vee \ D \ \vee \ t[b] \simeq s}$$

$$\frac{C \ \vee \ a \simeq b \qquad D \ \vee \ t[a] \not\simeq s}{C \ \vee \ \ D \ \vee \ t[b] \not\simeq s} \ \mapsto$$

$$\frac{C \ \vee \ a \simeq b \qquad \dfrac{a \simeq b \ \rightarrow t[a] \simeq t[b] \qquad D \ \vee \ t[a] \not\simeq s}{D \ \vee \ a \not\simeq b \ \vee \ t[b] \not\simeq s}}{C \ \vee \ D \ \vee \ t[b] \not\simeq s}$$

Suppose $t[a]$ is of the form $f(f(a))$, then we can apply congruence in two stages. First by introducing the clause

$$f(a) \simeq f(b) \ \rightarrow f(f(a)) \simeq f(f(b))$$

the second time the literal $f(a) \simeq f(b)$ is used in a conflict core, and then we introduce the implication

$$a \simeq b \ \rightarrow f(a) \simeq f(b)$$

This restricted way of applying congruence corresponds to the rule Dyn. Cong$^\sharp$.

Sup applied to an atomic disequality. The remaining Sup case we have not handled is inferences of the form:

$$\frac{C \ \vee \ a \simeq b \qquad D \ \vee \ a \not\simeq c}{C \ \vee \ \ D \ \vee \ b \not\simeq c}$$

There is no version of Dyn. Trans that allows learning lemmas of the form

$$a \simeq b \ \wedge \ a \not\simeq c \ \rightarrow \ b \not\simeq c$$

We show that this rule is not necessary, by examining the inference steps that involve $b \not\simeq c$ in the proof tree below. Let us first examine the case where the derivation has the form:

$$\dfrac{\dfrac{E \vee b \simeq d \qquad \dfrac{\dfrac{C \ \vee \ a \simeq b \qquad D \ \vee \ a \not\simeq c}{C \ \vee \ \ D \ \vee \ b \not\simeq c}}{\vdots}}{F \vee b \not\simeq c'}}{E \vee F \vee d \not\simeq c'}$$

In this case $a \simeq b \wedge a \not\simeq c \wedge b \simeq d \wedge \mathit{trail}\,(\Pi)_{\Pi'}$ is a theory conflict, where Π' is the derivation snippet given above. Rule Dyn. Trans lets us learn the clause $a \simeq b \wedge b \simeq d \rightarrow a \simeq d$, and we can rewrite the derivation to:

$$\dfrac{\dfrac{E \vee b \simeq d \qquad \dfrac{\dfrac{\dfrac{C \ \vee \ a \simeq b \qquad a \simeq b \wedge b \simeq d \rightarrow a \simeq d}{C \vee b \not\simeq d \vee a \simeq d} \qquad D \ \vee \ a \not\simeq c}{C \ \vee \ \ D \ \vee \ b \not\simeq d \vee d \not\simeq c}}{\vdots}}{F \ \vee \ b \not\simeq d \vee d \not\simeq c'}}{E \ \vee \ F \vee d \not\simeq c'}$$

The transformation is similar if the next inference that rewrites b modifies a proper sub-term of it.

E-Eqs also introduces a disequality literal. We can push applications of

this rule down in the derivation tree using the following transformation:

$$\dfrac{\dfrac{E \vee b \simeq d \qquad \dfrac{\dfrac{C \;\vee\; a \simeq b \vee a \simeq c}{C \;\vee\; a \simeq b \vee b \not\simeq c}}{\vdots}}{F \vee b \not\simeq c'}}{E \vee F \vee d \not\simeq c'}$$

$$\mapsto$$

$$\dfrac{\dfrac{\dfrac{\dfrac{E \vee b \simeq d \qquad a \simeq d \wedge b \simeq d \to a \simeq b}{E \vee a \simeq b \vee a \not\simeq d} \qquad \dfrac{E \vee b \simeq d \qquad C \vee a \simeq b \vee a \simeq c}{E \vee C \vee a \simeq d \vee a \simeq c}}{E \vee C \vee a \simeq d \vee d \not\simeq c}}{E \vee C \vee a \simeq b \vee d \not\simeq c}}{\vdots}}{E \vee F \vee d \not\simeq c'}$$

The transformation requires the clause $b \simeq d \wedge a \simeq d \to a \simeq b$. It can be learned using Dyn. Trans by first learning $b \simeq d \wedge a \simeq b \to a \simeq d$, as the original derivation ensures that $b \simeq d \wedge a \simeq b$ participate in a conflict. Then the helpful clause is learned from theory propagation on the assignment $b \simeq d \wedge a \simeq d$.

□

Lemma 2 makes use of Assumption 1. The derivation below introduces redundant literals $c \simeq d$ and $b \simeq d$.

$$\dfrac{\dfrac{a \simeq b \qquad \dfrac{b \simeq c \qquad \dfrac{a \simeq d \qquad c \simeq a}{c \simeq d}}{b \simeq d}}{a \simeq d} \qquad f(a) \not\simeq f(d)}{f(d) \not\simeq f(d)}$$

$$\bot$$

Our proof does not let us disregard such non-minimal derivations. Hence, the working assumption on the theory solver for equalities is that it admits arbitrary equality conflicts, including non-minimal ones. This is in contrast to how efficient congruence closure-based equality solvers work. They seek minimal conflicts. It is an open problem to strengthen the results to equality solvers that perform theory resolution based on a minimal unsatisfiable set of literals. We conjecture that CDER still p-simulates E-Res even when congruence produces only minimal conflicts.

1.11 Conclusions

We exemplified how modern SMT solvers combine algorithms for tractable sub-problems in a framework that addresses intractable problem domains. An important theme is to harness the size of the problem, since the search space grows exponentially in the problem size. A common trait in many methods was to delay polynomial space increase based on properties of the search and partial models (for model-based theory combination). In this context we examined a method for the *UF* theory that can find short proofs in many cases at the expense of introducing new literals. The new literals are introduced based on an analysis of theory conflicts. We took a proof theoretic perspective and compared the strength of the method relative to general unrestricted resolution proof systems. The main point is that the two proof systems are equally succinct.

Conflict Directed Theory Resolution is a general concept. The high-level point is to integrate theory resolution steps as part of the conflict analysis already performed by the DPLL(T) engine. We analyzed it only in the context of *UF*. With a reduction approach to theory solving, we noticed that A, D, CAL all reduce to *UF*, so an efficient *UF* solver is helpful in all these theories. It is fairly straight-forward to use the same ideas for conflict directed theory resolution for difference logic (example 2). The situation is more complex for LRA and especially LIA where also proof rules for cutting planes are needed [25].

References

[1] Michael Alekhnovich and Alexander A. Razborov. Resolution is not automatizable unless w[p] is tractable. *SIAM J. Comput.*, 38(4):1347–1363, 2008.

[2] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[3] Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura. Accelerating DPLL(T) using Joins - DPLL(⊔). In *Short papers at LPAR 08*, 2008.

[4] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in $UF(\mathcal{E})$. In *LPAR*, pages 557–571, 2006.

[5] Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 470–482. Springer-Verlag, July 1999.

[6] Randal E. Bryant, Steven German, and Miroslav N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[7] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize dll algorithms with clause learning. *CoRR*, abs/0811.1075, 2008.

[8] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages, 2nd GI Conference*, LNCS 33, pages 134–183, Kaiserslautern, West Germany, May 20–23, 1975. Springer-Verlag.

[9] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 1962.

[10] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT Solvers. In *CADE'07*. Springer-Verlag, 2007.

[11] Leonardo de Moura and Nikolaj Bjørner. Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[12] Leonardo de Moura and Nikolaj Bjørner. Model-based Theory Combination. *Electr. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*. Springer Verlag, 2008.

[14] Leonardo de Moura and Nikolaj Bjørner. Efficient, Generalized Array Decision Procedures. In *FMCAD*. IEEE, 2009.

[15] Leonardo de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 2010.

[16] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54:69–77, 2011.

[17] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[18] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[19] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[20] Bruno Dutertre and Leonardo de Moura. The Yices SMT Solver. http://yices.csl.sri.com/tool-paper.pdf, 2006.

[21] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV'04*, volume 3144 of *LNCS*, pages 175–188, 2004.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[23] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.

[24] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 283–290. AAAI Press, 2008.

[25] Dejan Jovanovic and Leonardo de Moura. Cutting to the chase solving linear integer arithmetic. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2011.

[26] Deepak Kapur and Calogero G. Zarba. A Reduction Approach to Decision Procedures. Technical Report TR-CS-1005-44, University of New Mexico, 2005.

[27] Lars Hörmander. The Analysis of Linear Partial Differential Operators II. *Grundlehren der mathematischen Wissenschaften*, 257, 1983.

[28] Karl Lieberherr. Complexity of superresolution. *Notices of the American Mathematical Society*, 24:A–433, 1977.

[29] Anatoli Ivanovic Malcev. Axiomatizable classes of locally free algebras of various types. In B. Wells III, editor, *The Metamathematics of Algebraic Systems*, volume 66, chapter 23, pages 262–281. North Holland, 1971. Collected Papers: 1936.1967.

[30] Yuri Matiyashevich. Hilbert's 10^{th} Problem: What can we do with Diophantine equations? In *A talk given at a seminar of IREM in Paris*, 2001.

[31] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[32] Orly Meir and Ofer Strichman. Yet another decision procedure for equality logic. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2005.

[33] Robert Nieuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007.

[34] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.

[35] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.

[36] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[37] Mirron Rozanov and Ofer Strichman. Generating minimum transitivity constraints in P-time for deciding equality logic. In *SMT 2007*, volume 198 of *ENTCS*, pages 3–17, 2007.

[38] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[39] Alexander Schrijver. *Combinatorial Optimization*, volume 24 of *Algorithms and Combinatorics*. Springer Verlag, 2003. In 3 volumes.

[40] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[41] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[42] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell.*, 9(2):135–196, 1977.