ORSAY
N° d'ordre :

UNIVERSITÉ DE PARIS-SUD

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD

PAR

Sylvain CONCHON

—×—

SUJET :

SMT Techniques and their Applications : from Alt-Ergo to Cubicle

soutenue le 11 décembre 2012 devant la commission d'examen

MM.	Gilles DOWEK	rapporteurs
	Stephan MERZ	
	Andrew TOLMACH	
Mme	Christine PAULIN	examinateurs
MM.	Joffroy BEAUQUIER	
	Giuseppe CASTAGNA	
	Silvio RANISE	

SMT Techniques and their Applications: from Alt-Ergo to Cubicle

Sylvain Conchon

Contents

Contents **v**

1 Introduction **1**
1.1 Automated Reasoning . . . 1
1.2 Deciding Union of Theories . . . 2
1.3 Combining Decision Procedures . . . 3
1.4 Satisfiability Modulo Theories . . . 4
1.5 My contributions . . . 5
1.6 Supports . . . 6

2 Combining Decision Procedures **7**
2.1 Motivations . . . 7
2.2 An Abstract Nelson-Oppen Framework . . . 8
2.3 Canonization for Disjoint Union of Theories . . . 12
2.4 Perspectives . . . 14

3 The Alt-Ergo SMT-solver **15**
3.1 Motivations . . . 15
3.2 Alt-Ergo's Input Language . . . 17
3.3 Typing and Lazy CNF . . . 24
3.4 The SAT Solver Module . . . 29
3.5 Matching . . . 33
3.6 The Equality Module . . . 34
3.7 The Arithmetic Module . . . 42
3.8 Perspectives . . . 45

4 The Cubicle SMT-based Model Checker **47**
4.1 Motivations . . . 47
4.2 Cubicle's Input Language . . . 47
4.3 Engineering an Efficient Reachability Modulo Theory Algorithm . . . 49

4.4 Perspectives 52

5 Conclusion and Perspectives **53**

Bibliography **57**

1
Introduction

My research activities are mainly focused on automated reasoning. More precisely, I'm interested in the development of the Satisfiability Modulo Theories (SMT) technology.

1.1 Automated Reasoning

At the crossroad of computer science and mathematical logic, automated reasoning is mostly dedicated to the use of computers for proving mathematical theorems.

What kind of theorems? Originally, most of the effort has been focused on the proof of algebraic or geometrical theorems. For instance, success stories of automated theorem provers (ATP) include the proof of the Robbins problem with EQP, results in quasi-groups and a range of algebraic structures with Otter, confirmation of Higman's lemma and Gerard's paradox with NUPRL.

Nowadays, ATP are mostly developed to prove logical formulas from formal verification of hardware and software. These formulas have some particularities. First, they are big in size (thousands of hypothesis and hundreds of axioms). Second, they involve a combination of several specific theory reasonings (equality, arithmetic, etc.). And third, they require a non-trivial treatment of universally or existentially quantified formulas.

While being simpler from a mathematical point of view, these formulas are difficult to prove with the generic ATP, also called TPTP provers, that have been successfully applied to prove fundamental mathematical theorems. The main reason for that is that handling such formulas requires an efficient mechanism for reasoning on a combination of specific theories. Unfortunately, it's still a challenge to integrate such mechanism in the uniform proof search procedures (e.g. resolution, tableaux) implemented in TPTP provers.

In parallel to the development of generic TPTP provers, several efficient decision procedures for specific theories have been designed. This includes propositional satisfiability solvers (SAT solvers), ground decision procedures for equality (congruence

closure), linear arithmetic (simplex) etc. The combination of these “little engine of proofs” is at the heart of the Satisfiability Modulo Theories technology.

1.2 Deciding Union of Theories

One of the main problem underlying the implementation of an SMT solver is the problem of deciding union of theories. This problem can be stated as follows.

If $\mathcal{T}_1$ and $\mathcal{T}_2$ are two *coherent* theories:

1. Is the union $\mathcal{T}_1 \cup \mathcal{T}_2$ coherent?
2. Can we build a decision procedure for $\mathcal{T}_1 \cup \mathcal{T}_2$ from decision procedures of $\mathcal{T}_1$ and $\mathcal{T}_2$?

While there is no answer in the general case, several results have been given for some classes of theories. For instance, when $\mathcal{T}_1$ and $\mathcal{T}_2$ have no symbol in common (*i.e.* when their signatures are disjoint), we have the following result due to Tinelli and Harandi [66] (Corollary 3.3 page 9).

$\mathcal{T}_1 \cup \mathcal{T}_2$ is coherent if $\mathcal{T}_1$ and $\mathcal{T}_2$ are disjoint and both admit an infinite model.

Once we know that the union of theories is coherent for a certain class of theories, we still have to find a way to combine the respective decision procedures of these theories. However, this is not possible in general as there exist unions of theories in which the constraint satisfiability problem is known to be undecidable while it is decidable in each component. Even in the simple case of *disjoint* theories, combining decision procedures is a non-trivial question. Let us see why on a simple example. Consider the combination technique that consists in deciding the satisfiability of a quantifier free formula Φ with symbols in two disjoint signatures Σ_1 and Σ_2 as follows:

1. Decompose Φ into two *pure* formulas Φ_1 and Φ_2 over Σ_1 and Σ_2, respectively.
2. Conclude that Φ is satisfiable if and only if Φ_1 and Φ_2 are proven to be satisfiable by using the decision procedures of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively.

This very naïve combination algorithm is not complete. For instance, let $\mathcal{T}_1$ be the theory of linear arithmetic and $\mathcal{T}_2$ the theory of equality with an uninterpreted function symbol f. Let Φ be the following conjunction of literals

$$(\Phi) \quad f(x) - x = 0 \wedge f(2x - f(x)) \neq x$$

The first step of the algorithm consists in a purification step: if a is a pure subterm of Φ that contains only symbols in Σ_i, then we replace a by a fresh variable z and we add the equation $z = a$ to Φ_i. After this variable abstraction mechanism, Φ is decomposed into two pure formulas Φ_1 and Φ_2 where:

$$
\begin{array}{ll}
(\Phi_1) & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\
(\Phi_2) & z_1 - x = 0 \wedge 2x - z_2 = z_3
\end{array}
$$

Both Φ_1 and Φ_2 are satisfiable in $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. However, the conjunction $\Phi_1 \wedge \Phi_2$ is not. Indeed, from $z_1 = f(x)$ and $z_2 = f(x)$, we deduce that $z_1 = z_2$. Now, $z_1 - x = 0$ implies $z_1 = x$, and from $2x - z_2 = z_3$ we deduce the equivalent class $z_1 = z_2 = z_3 = x$. Finally, by congruence, we deduce that $f(x) \neq z_2$ which contradicts $f(x) = z_2$.

This example illustrates an important property underlying the combination of decision procedures : the unsatisfiability of the conjunction $\Phi_1 \wedge \Phi_2$ is localized in a closed formula, called *interpolant*, that only contains symbols from the common signature $\Sigma_1 \cap \Sigma_2$. This result is due to Robinson and Craig [61]. Most algorithms for combining decision procedures are based on techniques for *efficiently* computing such interpolants.

1.3 Combining Decision Procedures

The basic design principles for combining decision procedures in the simple case of the union of disjoint theories have been set down thirty years ago in the landmark paper of Nelson and Oppen [54]. They described and proved a general combination algorithm which is at the heart of the Simplify [26] theorem prover.

The design of an efficient algorithm for combining decision procedures has been (and is still) an active domain of research and experimentations. For instance, at the same time as Nelson-Oppen algorithm was being designed, Shostak [62] proposed a combination algorithm for the theory of equality with uninterpreted symbols and specific equational theories, the so-called *Shostak theories*, for which there exist efficient procedures for, respectively, reducing terms to canonical form (*canonizers*) and turning equations into substitutions (*solvers*). Examples of such theories include the theories of linear arithmetic, pairs, bit-vectors etc.

This method interleaves canonization, equation solving and substitution application in a very tightly coupled way. For instance, in the previous example, the solver of linear arithmetic can be used to solve the first equation $f(x) - x = 0$. The resulting substitution $\sigma = \{f(x) \mapsto x\}$ is then applied to the rest of the formula $\sigma(f(2x - f(x))) \neq \sigma(x)$. This yields the literal $f(2x - x) \neq x$ which in turned is canonized to $f(x) \neq x$. Applying σ again gives the unsatisfiable literal $x \neq x$.

Shostak's method offered an apparently more efficient algorithm, but of restricted scope. What exactly the scope of Shostak's method is has remained unclear

for a long time [60, 5, 33, 41]. Furthermore, the method was based on the claim that the disjoint union of two (and therefore any finite number of) Shostak theories is a Shostak theory. However, the validity of this property received minimal serious attention and Shostak himself provided little evidence that this observation was correct. Nevertheless, the Shostak algorithm has influenced the design of several leading tools for automated verification, including PVS [57], SVC [4], and STeP [7]. It is also closely related to the ICS [31] decision procedure.

Proving correctness of such combination algorithms has always been an issue. For instance, it took twenty years to obtain the first correct versions of Shostak's algorithm [59]. Concerning the Nelson and Oppen algorithm, correctness becomes a concern as soon as we attempt to describe this framework at a lower level that explicates important implementation features. For instance, Barrett [3] verified a combination procedure with an impressive list of implementation features. The proof covers soundness but not termination, and takes over 120 pages.

1.4 Satisfiability Modulo Theories

The combination algorithms described in the previous section decide conjunctions of literals in a union of theories. In order to handle disjunctive formulas, most SMT solvers rely on efficient satisfiability procedures (SAT solvers) to perform case analysis. In particular, the DPLL/Davis-Putnam-Logemann-Loveland algorithm [22, 23] is at the heart of most SMT solvers.

DPLL is a backtracking search algorithm that tries to build a boolean model of a CNF formula. Starting from an empty Boolean assignment $\mathcal{A}$ and a set of clauses $\mathcal{S}$, DPLL incrementally builds a model by applying two main rules: the Boolean constraint propagation rule (BCP) and the splitting rule (SPLIT). SPLIT heuristically chooses an unassigned literal l and assigns a truth value to it in $\mathcal{A}$. BCP then deduces Boolean consequences for this assignment: clauses that contain l are removed from $\mathcal{S}$ and clauses of the form $\neg l \vee l_1 \vee \cdots l_n$ are reduced to $l_1 \vee \cdots \vee l_n$. When a clause is reduced to a single unassigned literal l, then there is no choice but to assign l to true and to recursively called BCP to deduce new consequences. If $\mathcal{S}$ becomes empty, then the initial formula is said to be satisfiable and $\mathcal{A}$ is a model. If a clause becomes empty after simplification, then DPLL must backtrack to the last splitting operation and try an opposite assignment.

DPLL has dramatically evolved over the last decade. Modern DPLL solvers, called CDCL/conflict driven clause learning solvers [64], can now handle problem with several million variables and clauses. There are at least three main improvements in CDCL solvers. First, they exploit conflict clauses (clauses that become empty after simplification) for backtracking to splitting rules other than the last one (backjuming) and to learn new interesting clauses [63]. Second, they implement an efficient indexing technique for detecting unit-clauses (two-watched literals [51]).

And third, they periodically restart the backtrack search while keeping the learning clauses [36].

At the same time CDCL solvers were making progress, the integration of SAT solvers and decision procedures for theories has received great attention. The DPLL($\mathcal{T}$) algorithm [56] describes the integration of DPLL/CDCL with a decision procedure for a theory $\mathcal{T}$. This integration is done in a three steps loop. First, every literals are replaced by Boolean variables. Second, the SAT solver is asked for a model $\mathcal{A}$. If there is no model, then the formula is unsatisfiable. Otherwise, $\mathcal{A}$ is converted back to theory literals and it is given to the decision procedure of $\mathcal{T}$. If $\mathcal{A}$ is consistent with $\mathcal{T}$, then the formula is $\mathcal{T}$-satisfiable. Otherwise, the third step of the loop consists in adding the clause $\neg\mathcal{A}$ as an explanation to the SAT solver which is then executed again.

Modern SMT solvers implement a tighter interaction between the SAT solver and the decision procedure for $\mathcal{T}$. For instance, BCP is interleaved with calls to the decision procedure to get theory models of partial assignments. Theory solvers are also extended with a theory propagation mechanism that help to deduce new fact implied by partial assignments at the theory level.

Only a few SMT solvers handle quantified formulas. They use usually an instantiation mechanism based on *triggers* [26, 52]. In this approach, the set of ground clauses handled by the SAT solver is periodically augmented by heuristically chosen instances of universally quantified formulas. The heuristic for choosing new instances is usually guided by patterns, also known as triggers, that are manually or automatically added to the universal formulas to restrict their instantiation to *known terms* (*i.e.* to terms that occurs in the partial assignment $\mathcal{A}$ of the SAT solver). This approach is not refutationally complete, even for pure first order logic. Other heuristic-based approach include a saturation process closed to the superposition calculus [53], model-based quantifier instantiation [34].

The SMT community has been very active for the last twenty years. Around forty SMT solvers have been developed and more than ten are still under active development [1]. The precursors tools in this domain are Simplify [26], the Shostak procedure in PVS [62], SVC [4], STeP [7] among others. State-of-the-art SMT solvers for first order logic include Z3 [25], CVC4 [6] and Yices2 [40].

1.5 My contributions

Most of my contributions in the automated reasoning domain are related to the design and implementation of the SMT solver Alt-Ergo and the SMT-based model checker Cubicle. Here is a resume of my results in that field:

– An abstract framework for combining decision procedures

1. see `http://smtlib.cs.uiowa.edu/solvers.html`

- Proofs about the combination of canonizers and the impossibility to combine Shostak solvers
- CC(X): a congruence-closure algorithm with solvable theories
- AC(X): ground AC completion with Shostak theories
- The formalization of a SAT solver with lazy CNF in Coq
- A simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic
- The implementation of the Alt-Ergo SMT solver
- The implementation of the Cubicle SMT-based model checker

The manuscript is organized as follows. Chapter 2 presents my contributions to the problem of combining decision procedures. Chapter 3 presents the Alt-Ergo SMT solver. Chapter 4 presents the Cubicle model checker. Chapter 5 contains a general conclusion.

This document presents only the main results of work. In particular, it contains no proof. References to papers containing more details are given in each chapter.

1.6 Supports

The development of Alt-Ergo and Cubicle has been (and is still) supported by several academic projects and industrial grants.

Academic Projects.

- ANR A3PAT (12/2005 - 06/2009). Design of CC(X) and first release of Alt-Ergo.
- ADT Alt-Ergo (2009-2011). Proof-Manager, a software to manage benchmarks; AltGr-Ergo, a graphical frontend to Alt-Ergo.
- ANR DECERT (01/2009 - 08/2012). A new decision procedure for the theory of linear arithmetic, and design of AC(X).
- FUI Hi-Lite (05/2010 - 05/2013). Generation of models and explanations.
- ANR BWare (09/2012 - 08/2016). Addition of rewriting capacities to the core of Alt-Ergo.
- ANR CAFEIN (01/2013 - 01/2016). Generation of interpolants.

Industrial activities.

- Airbus Grant (2011). Production of technical documents in view of the DO-178C qualification of Alt-Ergo.
- Intel Grant (2013). Algorithmic enhancements of Cubicle.

2
Combining Decision Procedures

This chapter presents an abstract framework for combining decision procedures and the results about the combination of canonizers and solvers of Shostak theories.

2.1 Motivations

I've started my research in automated deduction when I was a PostDoc at OGI School of Science and Engineering (Oregon Health and Science University, USA). Being interested both by implementation and correctness issues, I had undertaken the design of a Nelson-Oppen algorithm at a level that was high enough to enjoy a simple correctness proof (based on high-level results due to Tinelli and Harandi [66]), and low enough to incorporate crucial optimizations found in real implementations, like variable abstraction with sharing, theory state normalization, and deduction by lookup efficiently implemented by Shostak theory modules. With Sava Krstić, we came up with a combination framework, presented as a set of inference rules together with a simple strategy language capable of expressing complex combination algorithms, from the basic Nelson-Oppen to the very highly optimized one given by Shankar and Rueß [60]. Each algorithm is simply described by a regular expressions of inference rules. Proving correctness of a concrete algorithm written as a strategy in this framework amounts to proving one or two simply stated properties of the strategy; the rest follows from the correctness of the whole system.

I've also investigated the questions about the possibility of directly combining canonizers and solvers of Shostak theories. While almost all sources restated "the fact" that a canonizer for a disjoint union of theories was easy to obtain from canonizers of individual theories, no proof was given. It was also generally accepted that solvers cannot always be combined to produce a solver for the union theory. Convincing arguments for this were given in [60], but without reasonably complete proof. It was also often stated, e.g. in [5], that solvers for some Shostak theories do combine, but without proofs that this happens even for one pair of theories. With Sava Krstić, we attempted to understand and prove what can and what cannot be

combined. While reasonable definitions for a combination of two canonizers are not difficult to come up with, it is hardly self-evident that the "canonizers" they define satisfy the required properties. We proved that combining canonizers indeed goes as expected, assuming that the component theories are convex. The proof requires some effort, and simple counterexamples show that the convexity assumption would be difficult to relax. We also prove that under mild assumptions a disjoint union of theories *cannot* have a solver, regardless of the existence of solvers for the original theories.

The main results on these topics are presented in this chapter. More details can be found in [20, 43, 44, 21]

2.2 An Abstract Nelson-Oppen Framework

For the sake of simplicity, the system presented in this section only handles *convex* theories for which branching in unnecessary. I assume that the reader is familiar with basic notions of mathematical logic and model theory and refer for instance to [39, 61] for notations and conventions.

Following Nelson-Oppen's algorithm, our framework realizes the union $\mathcal{T}$ of disjoint *stably-infinite* theories $\mathcal{T}_i$, *i.e.* theories for which every quantifier-free formula satisfiable in some model of $\mathcal{T}_i$ is also satisfiable in an infinite model of $\mathcal{T}_i$. The abstract procedure is defined by a set of inference rules, shown in Figure 2.1. The rules describe the evolution of the state of the procedure, represented as a *configuration* $\langle V \parallel \Delta \parallel \Gamma \parallel \Phi_0, \ldots, \Phi_n \rangle$, where:

- Γ is a set of literals over $\mathcal{T}$; without lost of generality, we assume that Γ contains only equations or disequations, denoted by $a \bowtie b$;
- Δ is a set of (dis)equations between variables;
- each Φ_i is a set of equations of the form $x = a$ where x is a variable and a is a pure term of $\mathcal{T}_i$;
- V is a set of variables containing those occurring in Γ and Δ.

We write $\mathcal{C} \to \mathcal{C}'$, if a configuration $\mathcal{C}$ can be transformed into $\mathcal{C}'$ by applying one of the inference rules. The aim of our inference system is to determine satisfiability of configurations, defined as the satisfiability of the formula $\Gamma \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \wedge \Delta$, provided that the configuration $\bot$ is not satisfiable.

The rules $\mathbf{Abstract}_i$ are used to *purify* the literals of Γ. If a_π is a pure subterm of a at position π, then $\mathbf{Abstract}_i$ replaces a_π in a with a fresh variable z, at the same time adding the equation $z = a_\pi$ to the set Φ_i. The rules $\mathbf{Share}_i$ describe a space-efficient variable abstraction mechanism which allows us to replace a subterm a_π of a term a by an existing variable z which is known by one of the theories to be equal to a_π. The rule **Arrange** just transfers (dis)equations between variables from Γ to Δ. The rules $\mathbf{Contradict}_i$ and $\mathbf{Deduct}_i$ perform *equality propagation*

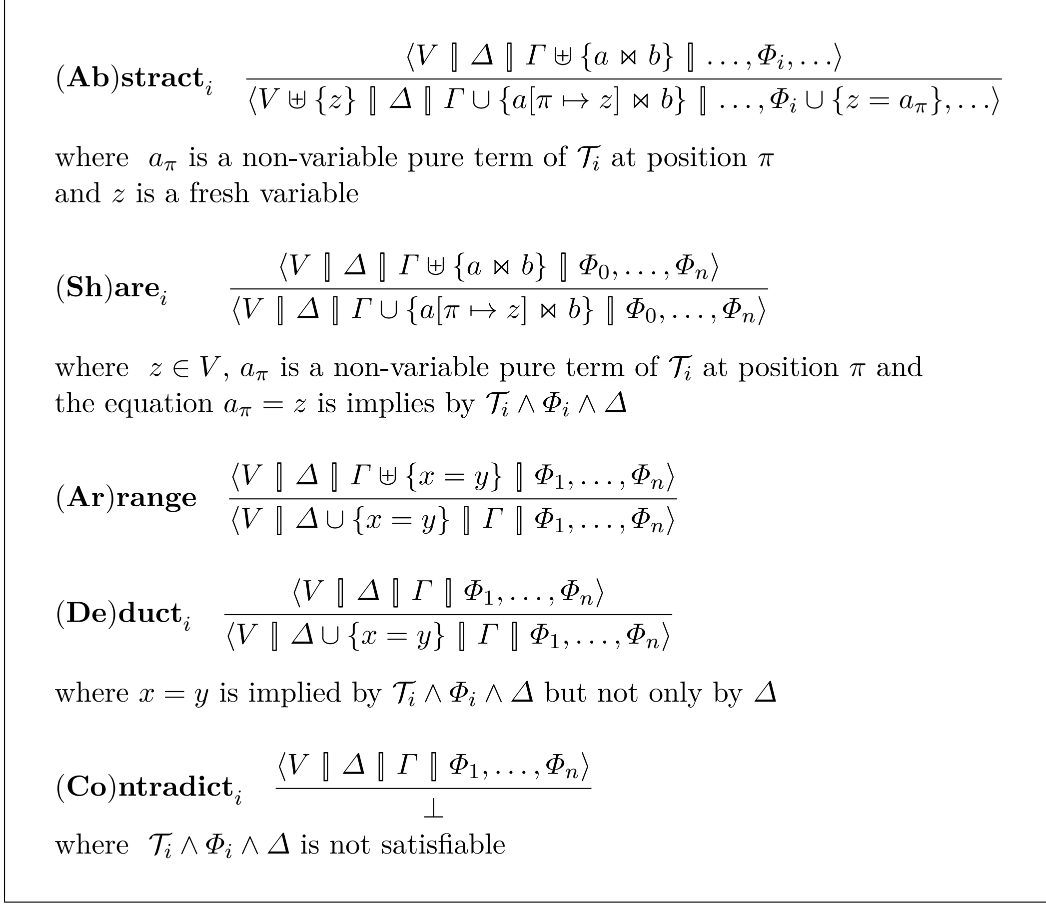

Figure 2.1: Inference system for combining decision procedures

by moving to Δ new equations between variables that are valid in some theory $\mathcal{T}_i$. The rule **Deduct**$_i$ adds to Δ a new equation that is not a logical consequence of Δ, but is possible to derive from Δ together with the "theory knowledge" Φ_i. The rule **Contradict**$_i$ produces a special configuration $\bot$ as soon as the state Φ_i becomes incompatible with Δ.

Theorem 1 (Correctness) *A set of formulas Γ is satisfiable if and only if there exists a configuration $\mathcal{C}$, distinct from $\bot$, such that $\langle V \parallel \varnothing \parallel \Gamma \parallel \varnothing \rangle \rightarrow^* \mathcal{C} \not\rightarrow$, where V is the set of variables in Γ.*

Shostak Optimizations

One difficulty when implementing a combination algorithm *à la* Nelson-Oppen is the rule $\mathbf{Deduct}_i$ which requires the generation of new equations that follow from the conjunction $\Phi_i \wedge \Delta$. In general, it is a non-trivial task to modify a decision procedure for such purpose. However, for some theories there exist efficient solutions to this problem. A prime example is the *free theory* with uninterpreted function symbols, where the congruence closure algorithm [55, 1] can process the input equations in Δ and change its state Φ_i accordingly so that new equations between variables can be directly seen from it. Shostak made an important discovery that a similar inference pattern is possible for many other theories [62]. Roughly speaking, the theory module maintains a union-find data structure on a set of terms so that the output equation $x = y$ is deduced by checking that $\mathsf{find}(x)$ is (syntactically) equivalent to $\mathsf{find}(y)$. To make such "trivial deduction" possible, the theory module must have some powerful mechanism for bringing its set of equations to some kind of normal form from which the maximum information about equalities between variables can be directly drawn.

We have formalized the pattern by a set of inference rules given in Figure 2.2. The first three rules are used to normalize a set of equations Φ_i so that it can simply be used as a union-find data-structure. $\mathbf{Subst}_i$ replaces every variable of a term by its representative (according to the equivalence classes generated by Δ). For $\mathbf{Canonize}_i$ to be applied, we assume that $\mathcal{T}_i$ is a convex theory equipped with *canonizer* that returns a unique normal form $\mathsf{canon}_i(a)$ for every term a. To complete the normalization process, $\mathbf{Solve}_i$ is used to realize the union operation between right hand side terms that have to be merged (by transitivity of equality). For that, we assume that $\mathcal{T}_i$ is equipped with a *solver* $\mathbf{solve}_i$ that takes an equation as input and returns its general solution in the form of an idempotent substitution, if the equation is satisfiable, and $\perp$ otherwise.

Finally, the rule $\mathbf{TDeduct}_i$ is a trivial special case of $\mathbf{Deduct}_i$, where the implied equality $x = y$ is found by a simple lookup into the state Φ_i. Similarly, $\mathbf{TShare}_i$ is a special case of $\mathbf{Share}_i$ that finds the required shared variable by inspecting the state.

Strategies

We introduce determinism in our inference system by constraining the shape of reduction chains. A variety of strategies can be described by using the simple language given in Figure 2.3. It is the language of regular expressions over the set of basic actions (rules of our inference system), extended with the operator $\oplus$. The figure also gives the semantics of the language: the concatenation $(\cdot)$, and choice $(+)$ operators have their standard meaning, the star $(*)$ is for exhaustive application, and $\oplus$ denotes a left-associative choice that gives preference to its left argument.

(Su)bst$_i$ $$\frac{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Phi_i \uplus \{x = a\}, \ldots\rangle}{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Phi_i \cup \{x = \Delta(a)\}, \ldots\rangle}$$
where $a \not\equiv \Delta(a)$

(Ca)nonize$_i$ $$\frac{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Phi_i \uplus \{x = a\}, \ldots\rangle}{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Phi_i \cup \{x = \mathsf{canon}_i(a)\}, \ldots\rangle}$$
where $a \not\equiv \mathsf{canon}_i(a)$

(So)lve$_i$ $$\frac{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Phi_i \cup \{x = a, y = b\}, \ldots\rangle}{\langle V \parallel \Delta \parallel \Gamma \parallel \ldots, \Psi \circ \Psi, \ldots\rangle}$$
where $\Delta(x) \equiv \Delta(y)$; $\mathsf{canon}_i(a) \not\equiv \mathsf{canon}_i(b)$; $a = b$ is $\mathcal{T}_i$-satisfiable and $\Psi = \Phi_i \cup \{x = a, y = b\} \cup \mathsf{solve}_i(a = b)$

(TDe)duct$_i$ $$\frac{\langle V \parallel \Delta \parallel \Gamma \parallel \Phi_0, \ldots, \Phi_n\rangle}{\langle V \parallel \Delta \cup \{x = y\} \parallel \Gamma \parallel \Phi_0, \ldots, \Phi_n\rangle}$$
where $\Delta(x) \neq \Delta(y)$; $\Phi_i(x) = \Phi_i(y)$

(TSh)are$_i$ $$\frac{\langle V \parallel \Delta \parallel \Gamma \uplus \{a \bowtie b\} \parallel \Phi_0, \ldots, \Phi_n\rangle}{\langle V \parallel \Delta \parallel \Gamma \cup \{a[\pi \mapsto z] \bowtie b\} \parallel \Phi_0, \ldots, \Phi_n\rangle}$$
where $a_\pi \in T_{\Sigma_i}(X) - X$; $\mathsf{canon}_i(\Delta(a_\pi)) = \Phi_i(z)$

Figure 2.2: Inference rules dedicated to Shostak theory modules

The following expressions describe different combination algorithms that can be expressed as strategies in our language. The first one represents the original Nelson-Oppen algorithm for the disjoint union of convex theories.

$$\mathbf{Ab}^* \cdot \mathbf{Ar}^* \cdot (\mathbf{Co} \oplus \mathbf{De})^* \tag{2.1}$$

where the action **Ab** is an abbreviation for $\mathbf{Ab}_0 + \cdots + \mathbf{Ab}_n$ and similarly **De** is the sum of all $\mathbf{De}_i$. The second expression describes an incremental version of the strategy (2.1) which processes one literal of Γ at a time.

$$((\mathbf{Va}^1 + \cdots + \mathbf{Va}^m) \cdot (\mathbf{Co} \oplus \mathbf{De})^*)^* \tag{2.2}$$

where $\mathbf{Va}^j$ is an abbreviation for the strategy $\mathbf{Ab}^* \cdot \mathbf{Ar}$ applied only to the jth literal of Γ. The third one optimizes the variable abstraction part of the previous strategies against proliferation of new variables by an aggressive use of the rules $\mathbf{Share}_i$.

$$a \ ::= \ \mathbf{Ab}_i \mid \mathbf{Ar} \mid \mathbf{Su}_i \mid \mathbf{Ca}_i \mid \mathbf{So} \mid \mathbf{Sh}_i \mid \mathbf{TSh}_i \mid \mathbf{De}_i \mid \mathbf{TDe}_i \mid \mathbf{Co}$$
$$e \ ::= \ a \mid e + e \mid e^* \mid e \cdot e \mid e \oplus e$$

$$\frac{\mathcal{C} \to \mathcal{C}' \quad \text{by applying the rule } a}{\mathcal{C} \to_a \mathcal{C}'}$$

$$\frac{\mathcal{C} \to_e \mathcal{C}' \qquad \mathcal{C}' \to_{e'} \mathcal{C}''}{\mathcal{C} \to_{e \cdot e'} \mathcal{C}''} \qquad \frac{\mathcal{C}_0 \to_e \cdots \to_e \mathcal{C}_n \not\to_e \qquad 0 \leq n}{\mathcal{C}_0 \to_{e^*} \mathcal{C}_n}$$

$$\frac{\mathcal{C} \to_e \mathcal{C}'}{\mathcal{C} \to_{e+e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \to_{e'} \mathcal{C}'}{\mathcal{C} \to_{e+e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \to_e \mathcal{C}'}{\mathcal{C} \to_{e \oplus e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \not\to_e \qquad \mathcal{C} \to_{e'} \mathcal{C}'}{\mathcal{C} \to_{e \oplus e'} \mathcal{C}'}$$

Figure 2.3: Syntax and semantics of a simple language for strategies.

$$(\mathbf{Sh} \oplus \mathbf{Ab})^* \cdot \mathbf{Ar}^* \cdot (\mathbf{Co} \oplus \mathbf{De})^* \tag{2.3}$$

Finally, the last expression describes with reasonable precision the highly efficient algorithm given by Shankar and Rueß [60]. We assume that the free theory is $\mathcal{T}_0$, and that $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are Shostak theories.

$$\Big(\mathbf{abstraction} \cdot (\mathbf{Co} \oplus \mathbf{merge} \oplus \mathbf{infer} \oplus \mathbf{normalize})^*\Big)^* \tag{2.4}$$

where

$$\begin{aligned} \mathbf{abstraction} &= (\mathbf{Va}^1 \oplus \cdots \oplus \mathbf{Va}^m) \cdot \mathbf{Su}_0^* \\ \mathbf{merge} &= (\mathbf{So}_1 \cdot \mathbf{Ca}_1^*) + \cdots + (\mathbf{So}_n \cdot \mathbf{Ca}_n^*) \\ \mathbf{infer} &= (\mathbf{TDe}_0 + \cdots + \mathbf{TDe}_n) \cdot \mathbf{Su}_0^* \\ \mathbf{normalize} &= (\mathbf{Su}_1 + \cdots + \mathbf{Su}_n) \cdot (\mathbf{Su}_1^* \cdots \mathbf{Su}_n^*) \end{aligned}$$

2.3 Canonization for Disjoint Union of Theories

I present in this section our results about canonization for disjoint union of theories. I shall only present the main results and refer to the full-version [44] for precise definitions, further explanations and technical details.

As mentioned in the previous section, a *canonizer* for a theory $\mathcal{T}_i$ is a function $\mathbf{canon}_i$ which, for a given term u in Σ_i returns a unique representative (the *canonical form*) of the $\mathcal{T}_i$-equivalence class of u. Thus, a computable canonizer solves the word problem for $\mathcal{T}_i$.

Given two pairwise disjoint theories $\mathcal{T}_1$ and $\mathcal{T}_2$ with respective signatures Σ_1 and Σ_2 and canonizers $\mathbf{canon}_1$ and $\mathbf{canon}_2$, we obtain a natural candidate canonizer

canon for $\mathcal{T}_1 \cup \mathcal{T}_2$ as the normal form function of a certain reduction system induced by canonizers $\mathbf{canon}_i$ on the set of mixed terms in $\Sigma_1 \cup \Sigma_2$.

To define such reduction system, we extend each canonizer $\mathbf{canon}_i$ to a function $\mathbf{ecanon}_i$ that can be applied on mixed terms (by treating its alien subterms as variables). These extended canonizers lead immediately to a reduction system $\rightarrow$ on the set of mixed terms as follows.

Definition 1 *Suppose π is a position in a mixed term t and suppose the root symbol of t_π is in Σ_i. If $\mathbf{ecanon}_i(t_\pi) \neq t_\pi$, we say that π is a* redex *of t and that t* reduces *to $t' = t[\pi \mapsto \mathbf{ecanon}_i(t_\pi)]$, symbolically $t \rightarrow t'$.*

This reduction system enjoys the following property which means that there is essentially only one generic way of using the canonizers of the component theories to fully reduce mixed terms. Let $\leftrightarrow^*$ the reflexive, symmetric and transitive closure of $\rightarrow$.

Theorem 2 *Every equivalence class of $\leftrightarrow^*$ contains exactly one irreducible term.*

The *candidate canonizer* **canon** induced by $\mathbf{canon}_1$ and $\mathbf{canon}_2$ is the function that maps every term t to its *normal form*—the unique irreducible term in the $\leftrightarrow^*$-equivalence class of t. A necessary and sufficient condition for **ecanon** to be a canonizer is given by the following result.

Theorem 3 *The function **canon** is a canonizer if and only if $u \neq v$ is $\mathcal{T}$-satisfiable for any two distinct irreducible terms u, v.*

The existence of two distinct irreducible mixed terms u, v such that $u = v$ holds in the union theory is indeed a necessary and sufficient condition for the failure of a candidate canonizer to be a canonizer. For a simple concrete example take the theory $\mathcal{T}$ with signature consisting of three constants p, q, r constrained by the axiom $p = q \vee p = r$, and take $\mathcal{T}'$ with one ternary function symbol f constrained by axioms $f(x, x, y) = f(x, x, x)$ and $f(x, y, x) = f(x, x, x)$. Then $f(p, q, r) = f(p, p, p)$ is a theorem of $\mathcal{T} \cup \mathcal{T}'$, while $f(p, q, r)$ and $f(p, p, p)$ are distinct irreducible terms. This condition is satisfied when the component theories are convex (and this is as much as we can hope for).

Theorem 4 *If $\mathcal{T}_1$ and $\mathcal{T}_2$ are convex theories then **canon** is a canonizer for $\mathcal{T}_1 \cup \mathcal{T}_2$.*

As an application of our result on combining canonizers, we prove that in general it is not possible to combine solvers. As mentioned in the previous section, a *solver* for a theory $\mathcal{T}_i$ is a function $\mathbf{solve}_i$ that takes an equation $u = v$ as argument, and returns a general solution for $u = v$ if this equation is satisfiable in $\mathcal{T}_i$. If $u = v$ is unsatisfiable, then $\mathsf{solve}(u = v)$ returns $\perp$.

In some trivial cases, it is possible to combine solvers. Suppose, for example, that $\mathcal{T}_1$ is a theory in which all function symbols are "projections" in the sense that $f(x_1, \ldots, x_n) = x_k$ holds in $\mathcal{T}_1$ for some k. It is not hard to see that then $\mathcal{T}_1 \cup \mathcal{T}_2$ has a solver for every theory $\mathcal{T}_2$ which has a solver. It turns out that these are pretty much all the cases when a combined theory allows a solver. Let us say that a function symbol f (of any non-zero arity) is *non-collapsing* when $f(x, \ldots, x) \neq x$ is satisfiable.

Theorem 5 *Suppose $\mathcal{T}_1$ and $\mathcal{T}_2$ are consistent stably-infinite theories with non-collapsing function symbols, and suppose* $\textbf{\textit{canon}}_1$ *and* $\textbf{\textit{canon}}_2$ *are canonizers of these theories. If the candidate canonizer* ***canon*** *defined by* $\textbf{\textit{canon}}_1$ *and* $\textbf{\textit{canon}}_2$ *is a canonizer for $\mathcal{T}_1 \cup \mathcal{T}_2$, then $\mathcal{T}_1 \cup \mathcal{T}_2$ does not have a solver.*

This is a strong negative result, at odds with claims that solvers of some common theories can be combined and at odds with implementations which apparently realize such combinations.

2.4 Perspectives

Alt-Ergo implements a combination algorithm for solvers of polymorphic typed theories. We have the strong feeling, but yet no proof, that combining solvers is possible for typed theories. We leave as future work the formalization and correctness proof of this algorithm.

3
The Alt-Ergo SMT-solver

This chapter presents Alt-Ergo, an open-source SMT solver under the CeCILL-C license. Alt-Ergo is available at `http://alt-ergo.lri.fr`.

3.1 Motivations

After my PostDoc, I joined the Démons team (now called Toccata) at University Paris-Sud. This team develops tools for program verification. In that specific domain, the goals to be proved are usually given as formulae in a polymorphic multi-sorted first-order logic. Since at that time SMT solvers were not able to deal directly with polymorphic types, we decided, Évelyne Contejean and myself, to develop a new SMT solver called Alt-Ergo dedicated to the resolution of polymorphic and multi-sorted proof obligations. Nowadays, Alt-Ergo is still the only existing SMT solver dealing directly with parametric polymorphism.

Before handling polymorphic formulas, we first had to design the core of our new SMT solver. Following my work on the combination of decision procedures and, in particular, on the optimized Shostak's method, I realized that the combination of the free theory of equality $\mathcal{E}$ with other theories plays a central role in any SMT solver (this is clearly reflected nowadays in all categories of the SMT competition). More precisely, I was convinced that an efficient combination of $\mathcal{E}$ with the theory of arithmetic $\mathcal{A}$ should be at the core of Alt-Ergo. Since Shostak's method has been specifically designed for combining $\mathcal{E}$ with solvable and canonizable theories (such as $\mathcal{A}$), I decided to use it at the core of Alt-Ergo. However, a central point for Shostak method to be effective is that it has to handle *terms*. As a consequence, the main operations of the algorithm, substitution application, normal form reduction and equation resolution, have to be directly implemented on term data structures, which is not the best efficient way of implementing a decision procedure (*e.g.* a term data structure is obviously not optimal to manipulate polynomials). However, relaxing this constraint has strong impacts on the design of the method. Indeed, bringing a term into a normal form amounts to traversing its *syntactic structure* for

applying the canonizers on interpreted subterms. This *global canonization* is at the heart of the method and it also guarantees the incrementality of the algorithm.

We thus decided to start the development of a new algorithm, called CC(X) (for congruence closure modulo X), which combines the theory $\mathcal{E}$ with an arbitrary built-in solvable theory X without using canonizers. This algorithm uses *abstract values* as representatives allowing efficient data structures for the implementation of solvers. CC(X) is presented as a set of inference rules whose description is low-level enough to truly reflect the actual implementation of the combination mechanism in Alt-Ergo. It also enforces to entirely rebuild the algorithm since global canonization, which is at the heart of Shostak combination, is no longer feasible with semantic values. The main results on this topics are presented in Section 3.6 and in more details in [15, 16].

Concerning the theory of arithmetic, we concentrated our efforts on the linear fragment for the *integers* which is intensively used in program verification. We came up with a novel decision procedure which tries to bridge the gap between projection based techniques (*e.g.* Omega-Test) and branching/cutting methods (*e.g.* branch-and-bound, branch-and-cut, Gomory cuts). Our decision procedure interleaves an exhaustive search for a model with bounds inference. These bounds are computed provided an oracle capable of finding constant positive linear combinations of affine forms. Currently, the distributed version of Alt-Ergo uses the Fourier-Motzkin algorithm to find such bounds. But we also have designed a prototype with an efficient oracle based on the Simplex procedure for which experimental results show that our approach is competitive with state-of-the-art SMT solvers. The main results on this topics are presented in Section 3.7 and in more details in [11].

When we started to address the problem of handling quantified formulas, we rapidly faced a problem specific to SMT solvers : the conversion to clausal form required by the SAT engine of Alt-Ergo strongly damages the performance of the instantiation-based mechanism used to handle quantifiers. We found a simple and elegant solution to this issue: the lazy CNF conversion scheme. To prevent the exponential blow-up in size of the CNF conversion, our lazy mechanism introduces a notion of *lazy literals* which can be expanded in a call-by-need fashion by the DPLL procedure. Contrary to the traditional Tseitin variables, lazy literals do not disrupt the SAT solver algorithm and do not add irrelevant ground terms during the proof search that can be used to generate useless instances of lemmas. For efficiency reasons, we have implemented lazy literals using a parametric hash-consing library that safely ensures maximal sharing by typing. Moreover, the parameters used for the equality relation and the hashing function attempt to identify some subformulas that are logically equivalent. This generally allows more subformulas to be structurally shared which can give a big performance boost to the procedure. The main results on this topics are presented in Sections 3.3 and 3.4 and in more details in [28, 47, 48].

Surprisingly, only a small number of modifications were needed to bring polymorphic types to Alt-Ergo. The first one occurred in the typing module where unification was necessary for solving polymorphic constraints over types. The second one consisted in extending triggers' definition of the instantiation mechanism in order to deal with both term and type variables. Last, the matching module had to be modified to account for the instantiation of type variables. An important consequence demonstrated by our work is that no other modification is required on an SMT solver to handle polymorphic types. The main results on this topics are presented in Section 3.5 and in more details in [8].

Very often, proof obligations coming from program verification contain axioms that are just purely administrative : they represent basic properties such as associativity and commutativity (AC), transitive closures, etc. Contrary to other lemmas, a huge number of instances of administrative lemmas can be necessary to prove a goal. Unfortunately, the mere addition of such axioms in a proof obligation will usually glut the solver with plenty of useless equalities which will strongly impact its performances. To work around the specific problem of handling AC properties, we have designed a modular extension of ground AC-completion for deciding formulas in the combination of the theory of equality with user-defined AC symbols, uninterpreted symbols and an arbitrary signature disjoint Shostak theory X. Our algorithm, called AC(X), is obtained by augmenting in a modular way ground AC-completion with the canonizer and solver present for the theory X. This integration rests on canonized rewriting, a new relation reminiscent to normalized rewriting, which integrates canonizers in rewriting steps. The main results on this topics are presented in Section 3.6 and in more details in [12, 13, 14].

3.2 Alt-Ergo's Input Language

The input language of Alt-Ergo is a first-order logic with some built-in theories and polymorphic data types. More precisely, Alt-Ergo's input language includes:

- propositional connectives $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$;
- quantifiers $\forall$ and $\exists$, and user-defined triggers;
- user-defined polymorphic and enumerative data types;
- the theory of the equality symbol = with uninterpreted function symbols;
- the theory of arithmetic over the integers and the rational which defines the function symbols $+, -, \times, /,$ mod and the relation symbols $<, >, \leq, \geq$
- the theory of user-defined associative and commutative symbols;
- the theory of polymorphic functional arrays with accessors and updates;
- the theory of user-defined polymorphic records.

Propositional Logic. Alt-Ergo has a built-in data type `prop` for propositional variables. Its propositional connectives are `and`, `or`, `->`, `<->` and `not`.

```
logic p, q, r, s: prop
goal g: ((p -> (q and r and s)) and not s) -> not p
```

Predicates are declared as functions with a co-domain of type `prop` and `axiom` declarations are used to state facts.

```
type person logic david, nancy: person

logic man: person -> prop
logic woman:person -> prop

axiom facts : man(david) and woman(nancy)
```

Alternatively, a predicate can be defined directly by using a `predicate` declaration which combines its declaration and its definition.

```
predicate man_or_woman(p:person) = man(p) or woman(p)
goal g: man_or_woman(david)
```

Quantifiers and Triggers Alt-Ergo has some supports for quantified formulas. It can handle the universal quantifier `forall` and the existential quantifier `exists`.

```
logic father_of: person, person -> prop
axiom father_man: forall e,p:person. father_of(p,e) -> man(p)
```

Predicates' arguments are implicitly universally quantified.

```
logic mother_of: person, person -> prop
predicate son_of(e:person, p:person) =
  father_of(p,e) or mother_of(p,e)
predicate grandfather_of(gp:person, e:person) =
  exists p:person.
     father_of(gp,p) and ( father_of(p,e) or mother_of(p,e))

goal g1:
  forall gp,p:person. grandfather_of(gp,p) -> man(gp)

goal g2:
  forall g,p:person.
    grandfather_of(g,p) <->
      exists pp:person. father_of(g, pp) and son_of(p,pp)
```

Alt-Ergo handles universal formulas through an instantiation mechanism. The heuristic for choosing new instances is based on *triggers*. A trigger is a list of patterns that restrict instantiation to known (ground) terms that have a given form. For instance, if `P(x)` is used as an instantiation pattern for the following axiom `ax1`

```
logic P,Q,R : int -> prop
axiom ax1 : forall x:int. (P(x) or Q(x)) -> R(x)

goal g3 : P(1) -> R(1)
goal g4 : Q(2) -> R(2)
```

then, among the set {`P(1)`,`R(1)`,`Q(2)`,`R(2)`} of known terms, only `P(1)` can be used by the matching algorithm to create the instance `((P(1) or Q(1)) -> R(1)` of `ax1`, which implies that only `g3` is proved.

Alt-Ergo's input language provides also the possibility for the user to specify its own triggers. For instance, the list of terms `[f(x), Q(y)]` is an explicit trigger for axiom `ax2`.

```
logic P,Q,R : int -> prop
logic f : int -> int
axiom ax2 : forall x,y:int [f(x), Q(y)]. P(f(x)) and Q(y) -> R(x)
```

It is also interesting to note that substitutions are built modulo equality. For instance, even if no ground terms match the pattern `P(f(x))` in the following problem

```
logic P,R : int -> prop
logic f : int -> int
axiom ax : forall x:int [P(f(x))]. P(f(x)) -> R(x)
goal g1 : forall a:int. P(a) -> a = f(2) -> R(2)
```

the instance `P(f(2)) -> R(2)` of axiom `ax` is generated by combining the ground equality `a = f(2)` and the ground term `P(a)`.

Polymorphic and Enumerative Data Types. The main features of Alt-Ergo *w.r.t.* other SMT solvers is its polymorphic logic. Polymorphic types can appear in built-in theories or in user-defined data types. Alt-Ergo has a type discipline *à la* ML, which means type variables are implicitly quantified by prenex universal quantifiers.

```
type 'a list
logic nil: 'a list
logic cons: 'a, 'a list -> 'a list
```

```
logic hd: 'a list -> 'a
logic tl: 'a list -> 'a list

axiom construction :
  forall l:'a list. l<> nil -> cons(hd(l),tl(l)) = l
axiom hd_cons :
  forall x:'a. forall l:'a list. hd(cons(x,l)) = x
axiom tl_cons :
  forall x:'a. forall l:'a list. tl(cons(x,l)) = l

goal g :
  forall l:'a list. forall r:('a list) list.
  r<>nil -> l = hd(r) -> l<>nil ->
  cons(cons(hd(l),tl(l)), tl(r)) = r
```

Alt-Ergo provides also a theory of polymorphic records and a support for user-defined enumerative data types.

```
type choice = A | B | C
type 'a t = { a : 'a; b : int }

goal g :
 forall r:choice t.
    r.a <> A and r.a <> B -> { r with a = C} = r
```

The Theory of Free Equality. Alt-Ergo's theory of free equality defines the semantic of equality with uninterpreted function symbols. In this theory, the signature of the equality symbol = is `'a -> 'a -> prop`.

```
type t
logic h: 'a -> 'a
logic g: 'a,'a -> 'a

goal g:  forall a,x:t. h(x)=x and g(a,x)=a -> g(g(a,h(x)),x)=a
```

Alt-Ergo also provides two built-in predicates `<>` and `distinct` for disequalities. The semantic of `x<>y` is `not(x = y)`, while the semantic of `disctinct(x,y,z)` is `not(x = y) and not(y = z) and not(x = z)`.

```
logic x,y,z,t : int
logic f : int, int -> int
```

```
goal g1 : y = t -> f(x,y) <> f(x,z) -> t <> z
goal g2 : forall a,b,c : 'a. distinct(a,b,c) -> a <> c
```

The Theory of Arithmetic. Arithmetic expressions are of the following form:

$$e \quad := \quad t \mid \mathtt{n} \mid e\mathtt{+}e \mid e\mathtt{-}e \mid e\mathtt{*}e \mid e\mathtt{/}e \mid e\mathtt{\%}e \mid e\mathtt{<}e \mid e\mathtt{>}e \mid e\mathtt{<=}e \mid e\mathtt{>=}e$$

where `n` is an integer or a rational constant and t is either a variable or a non-arithmetic *term*. The arithmetic operators supported by Alt-Ergo are addition `+`, subtraction `-`, multiplication `*`, division `/` or modulo `\%`. Alt-Ergo proposes two built-in types for numerical values: `int`, for values in $\mathbb{Z}$, and `real` for the values in $\mathbb{Q}$. Arithmetic expressions are typed by the following typing rules:

– if `op` $\in \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{/}\}$ and e_1 and e_2 are two arithmetic expressions of type `int` (resp. `real`), then e_1 `op` e_2 is of type `int` (resp. `real`);
– if e_1 and e_2 are of type `int`, then $e_1\%e_2$ (e_1 modulo e_2) is of type `int`.
– If `op` $\in \{\mathtt{<}, \mathtt{>}, \mathtt{<=}, \mathtt{>=}\}$ and e_1 and e_2 are two arithmetic expressions of type `int` (resp. `real`), then e_1`op` e_2 is of type `prop`.

It's important to note that only types can distinguish between division on the integers (Euclidean division) and division on the reals.

Alt-Ergo implements a Shostak theory module for linear arithmetic and an extension of Fourier-Motzkin decision procedure with interval arithmetic to fully handle integer linear expressions. For instance, Alt-Ergo can prove the following formula:

```
goal g1 :
  forall x,y: int.
     2 * y - x <= 0 and -8 * y + x + 2 <= 0 and 2 * y + x - 3 <= 0
     -> false
```

Non-linear expressions can also be handled, but in a very limited way. For instance, Alt-Ergo tries to systematically develop non-linear terms.

```
goal g2 : forall x,y:int. x * (x + 1) = y -> x * x = y - x
```

Its treatment of expressions like e_1/e_2 and $e_1\%e_2$ is essentially limited to the case where e_2 is a constant, or when e_1 is a multiple of e_2

```
goal g3: forall a:int. a % 8 = 6 -> a % 4 = 2 and a % 2 = 0
goal g4 :
   forall x,y:int.
    (x * x + y * (2 * x + y)) <>0 ->
    (2 * x * x + 4 * x * y + 2 * y * y) / ( (x + y) * (x + y)) = 2
```

Interval arithmetic is also extended to handle non-linear expressions.

```
goal g5 :
  forall x,y:int.
   2 <= x <= 6 and -3 <= y < 0 ->
   -84 <= 3 * x + 2 * y + 4 * x * y + 2 * (x / y) <= 6

goal g6 :
  forall x,y:int.
   2 <= x <= 6 and -3 <= y < 0 ->
   -64 <= 3 * x + 2 * y + 4 * x * y + 2 * (x / y) <= -8

goal g7 :
  forall x:int.
   2 <= x * x <= 9 ->  x= -2 or x = -3 or 2 <= x <= 3

goal g8: forall x,y:int. x * x * x = -1 -> x + y = 4 -> y = 5
```

It is also important to note that the division operator is total in Alt-Ergo, in particular division by zero is uninterpreted.

The arithmetic module cooperates with the theory of AC symbols (see below) to extend the treatment of non-linear terms. Alt-Ergo can thus prove some formulas just by considering the associative and commutative property of the `*` operator.

```
goal g :
   forall v,t,w:int. v * t = 3 and v * w = 5 -> 3 * w = 5 * t
```

This cooperation can also help to convert non-linear terms into linear terms.

```
goal g :
   forall x,v,t,w:int.
     v * t = 3 and v * w = 6 ->
     (w - 2 * t + 2) * x = 8 ->
     x = 4
```

In that example, the AC theory deduces `w = 2 * t` from `v * t = 3` and `v * w = 6` so that the non-linear term `(w - 2 * t + 2) * x` is thus converted into `2 * x`.

The Theory of Associative and Commutative Symbols. Alt-Ergo implements a decision procedure for user-defined AC symbols. The declaration of binary AC symbols in done with a `logic ac` statement.

```
logic ac u : int, int -> int

goal g1 : u(1,u(2,u(3,u(4,5)))) = u(u(u(u(5,4),3),2),1)

goal g2 :
  forall a,b,c,v,w:int.
   u(a,b) = w and u(a,c) = v -> u(b,v) = u(c,w)
```

The Theory of Polymorphic Arrays. This theory provides a built-in type `('a,'b) farray` and a built-in syntax for manipulating polymorphic functional arrays.

```
goal g:
  forall i,j:int.
  forall r:(int,int) farray.
  forall m:(int,(int,int) farray) farray.
  r = m[i] -> m[i<-r[j<-r[j]]] = m
```

Given an array `a`, an index `i` and a value `v`, the access operation is written `a[i]` and the update operation is `a[i <- v]`.

Alt-Ergo's Architecture. The overall architecture of Alt-Ergo is represented in Figure 3.1. It is made of two parts : the *front-end* and the *main loop*. The front-end contains two parsers (SMT-Lib format and Alt-Ergo's native syntax), a type checker and a module for preparing the treatment of quantified formulas. The main loop is composed of a SAT solver, a matching module for handling quantifiers and triggers, and a set of built-in decision procedures.

The SAT solver module takes the role of the conductor. It handles the propositional structure of formulas, sends ground literals to the decision procedures and calls the matching module to create instances of quantified formulas. The other modules code can be classified into three categories:

1. The first category contains the modules `CC(X)`, `UF(X)` and `AC(X)`. The first two handle the theory of equality while the third one implements the AC theory. More precisely, `CC(X)` is a variant of the Shostak algorithm [16] that uses a generic union-find data structure implemented in `UF(X)`, and `AC(X)` is based on an extension of ground AC completion [13]. For each module, the parameter `X` represents a Shostak theory extended with an internal state used to handle built-in predicates.
2. The second category implements a combination algorithm that "builds" a single theory `X` from a set of decision procedures.

3. The last category contains the decision procedures for the theories of arithmetic, arrays, records and enumeration types. In particular, the arithmetic module is composed of three sub-modules: `Arith` contains Shostak solvers, `FM` and `Intervals` implement Fourier-Motzkin and interval arithmetic, respectively.

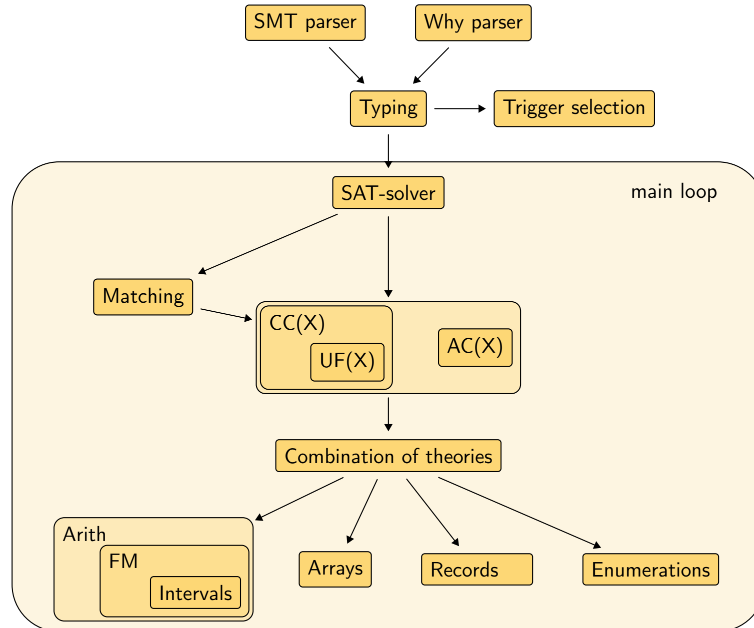

Figure 3.1: Alt-Ergo's architecture

In the rest of this chapter, I will describe several modules of this architecture. In particular, I will show the impact of adding polymorphic types in the typing module as well as the matching module. I will also present the implementation of the SAT solver and the equality modules (`CC(X)`, `UF(X)` and `AC(X)`) by a set of inference rules that are directly extracted from the source code. Finally, I will give some detail about the theory module for arithmetic.

3.3 Typing and Lazy CNF

The type system of Alt-Ergo is very similar to the type system of a simple functional programming language. However, the interaction between polymorphic types and first-order formulas introduce some subtleties. Discussing this point will lead us to give an intuitive semantic for our polymorphic first-order logic. The typing module is also responsible for computing triggers and converting input formulas in

conjunctive normal forms. In order to prevent the well-known exponential blow-up in size of this transformation, Alt-Ergo implements a lazy CNF conversion scheme which, unlike Tseitin-style approaches, does not disrupt the lemma instantiation mechanism. The material presented in this section is discussed in [48, 28, 8].

Polymorphism and its subtleties. Up to to now, I remained vague about the meaning of the free type variables in the definition of parametric symbols and polymorphic axioms. In order to highlight the subtleties introduced by polymorphism, I will now *explicitly* denote the *implicit* quantification of these type variables by using the symbol ∀ and type variables instantiation (in terms) by brackets. With these new conventions, the beginning of the axiomatization of the list theory becomes (for the sake of readability, I shall replace verbatim type variables `'a`, `'b`, ... by Greek letters $\alpha, \beta, ...$):

```
type ∀α. α list
logic cons:∀α. α, α list -> α list
logic hd:∀α. α list -> α

axiom hd_cons :
    ∀α. forall x:α. forall l:α list. hd[α](cons[α](x,l)) = x
```

The type `list` can be understood as a type family, *i.e.* a function yielding one `list` type for each type α. Similarly, the declaration of a logical symbol like

```
logic hd:∀α. α list -> α
```

introduces a function family: for each type τ, it provides a function $\texttt{hd}_{[\tau]}$ of type τ `list -> ` τ. The case of axiom `hd_cons` is more informative: the type variable in this axiom is universally quantified at the outer level of the definition. This outermost type quantification is general in Alt-Ergo and reflects our choice of prenex-polymorphism *à la* ML [50]. A very important consequence of this fact is that an axiom is different from an hypothesis in a goal. This can be seen in the following example:

```
type ∀α. α t
logic P:∀α. α t -> prop
axiom ax1:∀γ. forall x:γ t. P[γ](x)
goal g1:∀α,β. (forall x:α t. P[α](x)) and (forall x:β t. P[β](x))
```

Th goal `g1` can be easily proved by Alt-Ergo by instantiating axiom `ax1` once for each type variable α and β. Now consider putting the axiom `ax1` as an hypothesis in the goal:

```
goal g2:.
  ∀α,β,γ.
    (forall x:γ t. P[γ](x)) ->
    (forall x:α t.  P[α](x)) and (forall x:β t. P[β](x))
```

The goal g2 is not valid anymore since it is only provable when α, β and γ are equal. This is a manifestation of the fact that in general the formulas $(\forall x.P \Rightarrow Q)$ and $(\forall x.P) \Rightarrow Q$ are not equivalent. Note, by the way, that even if the goal g2 is polymorphic, a goal can always be considered monomorphic [1]: for every type variable α universally quantified in the goal, it suffices to introduce a fresh ground type t_α and to substitute α by t_α. Proving this particular instantiation of the goal is equivalent to proving it for any instantiation, since no assumptions are made on the fresh types t_α.

The theory of polymorphic lists illustrates also an interesting phenomenon. Consider the declaration of the nil symbol:

```
logic nil:∀α. α list
```

The symbol nil is a so-called *polymorphic constant*, that is a family of constants, one for each type. This implies that for each type α, the type α list is inhabited by $\texttt{nil}_{[\alpha]}$, even if α is not! A more pernicious declaration is

```
logic any:∀α. α
```

which makes *every* type inhabited. Polymorphic constants will have to be dealt with carefully during triggers' definition and matching (*cf.* Section 3.5).

We have now gained enough understanding about what parametric polymorphism means to be able to give a good intuition of the semantics of polymorphic first-order logic. Suppose we have a polymorphic theory $\mathbf{T}_{\text{PFOL}}$ and a goal $\mathbf{G}$; we wish to explain what it means for $\mathbf{T}_{\text{PFOL}}$ to *entail* $\mathbf{G}$, in symbols $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G}$. As argued above, we can consider that $\mathbf{G}$ is monomorphic without restriction. Let $\mathcal{T}$ be the set of all ground types that can be built from the signature in $\mathbf{T}_{\text{PFOL}}$ along with an infinitely countable set of arbitrary fresh constant types [2]. Now, let us write $\mathbf{T}_{\text{FOL}}$ the monomorphic multi-sorted theory such that:

- the set of its types is $\mathcal{T}$;
- its function symbols are all monomorphic instances (with types in $\mathcal{T}$) of the function symbols defined in $\mathbf{T}_{\text{PFOL}}$;
- similarly, its axioms are all the possible monomorphic instances of the axioms in $\mathbf{T}_{\text{PFOL}}$.

Given such a theory, we have that $\mathbf{T}_{\text{PFOL}}$ entails $\mathbf{G}$ if and only if $\mathbf{T}_{\text{FOL}}$ entails $\mathbf{G}$ in monomorphic first-order logic, ie. $\mathbf{T}_{\text{PFOL}} \models_{\text{PFOL}} \mathbf{G} \Leftrightarrow \mathbf{T}_{\text{FOL}} \models_{\text{FOL}} \mathbf{G}$. Further-

1. This means that pushing a polymorphic axiom into a goal makes it monomorphic.
2. This technical requirement ensures that $\mathcal{T}$ be non-empty and that polymorphic goals can be monomorphized.

more, since the proof of $\mathbf{G}$ in $\mathbf{T}_{\text{FOL}}$ is finite, only a finite number of monomorphic instances of the definitions in $\mathbf{T}_{\text{PFOL}}$ are necessary to establish a proof of $\mathbf{G}$. Altogether, this means that the task of solving a polymorphic problem amounts to finding the right monomorphic instances of the definitions in the problem and then solving the monomorphic problem we obtain in this manner. The task of generating and finding monomorphic instances is discussed in Sections 3.5.

Triggers and Polymorphism Triggers are computed when typing formulas. If no explicit trigger is provided, Alt-Ergo chooses the longest term of the formula that contains all quantified variables. For instance, the selected trigger for the axiom `ax` bellow is `R(f(f(x)))`

```
logic P,R : int -> prop
logic f : int -> int
axiom ax : forall x:int. P(f(x)) -> R(f(f(x)))
```

When no such term exists, Alt-Ergo selects a list of terms. For instance, the selected trigger for the following axiom is the list `[P(f(x)), Q(y,z)]`.

```
logic P,R : int -> prop
logic Q: int, int -> prop
logic f : int -> int
axiom ax :
 forall x,y,z:int. P(f(x)) and P(z) and Q(y,z) -> R(f(f(z)))
```

Polymorphism has also some subtle interaction when computing triggers. To be well-formed, triggers have to cover all variables of a formula, term variables and *type* variables. For instance, in the following example,

```
type ∀α. α t
logic P, Q : ∀α. α t -> prop
axiom ax2:
  ∀α. forall x:int t. P(x) -> (forall y:α t. P(y) -> Q(y)) -> Q(x)
```

triggers for axiom `ax2` should not only contain the term variable `x`, but also the type variable α. Unfortunately, there is no such term and Alt-Ergo fails in computing triggers for this formula.

Triggers may also only contain type variables. For instance, the trigger for the following axiom is the ground term `length(nil)`

```
axiom length_nil:  ∀α. length(nil) = 0
```

This example illustrates a side effect of polymorphism : a ground fact in untyped first-order logic may become a (quantified) axiom in polymorphic first-order logic.

Lazy CNF. The SAT solver of Alt-Ergo requires input formulas to be converted in CNF. In Alt-Ergo, this conversion is done during type checking and benefits from the hash-consing mechanism aggressively used in Alt-Ergo for building terms, literals, formulas etc.

There are different well-known techniques for implementing CNF conversion. The first possibility is to do a naive, traditional, conversion that uses de Morgan laws in order to push negations through the formula to the atoms' level, and distributes disjunctions over conjunctions until the formula is in CNF. For instance, this method would transform the formula $R \vee (P \wedge Q)$ in $(R \vee P) \wedge (R \vee Q)$. It is well-known that the resulting formula can be exponentially bigger than the original.

A well-known solution to the CNF blow-up is due to Tseitin [67]. It consists in introducing *proxy-variables* (proxies for short) for all subformulas of the input formula F and to initialize the solver with all the clauses defining these proxies plus the proxy variable for F. The set of clauses thus obtained is *equisatisfiable* to the original formula. For instance, introducing a proxy φ for a sub-formula $P \wedge Q$ (where P and Q are propositional variables) amounts to introducing the three *proxy-clauses* (pclauses for short) $\neg\varphi \vee P$, $\neg\varphi \vee Q$ and $\varphi \vee \bar{P} \vee \bar{Q}$ whose conjuction is equivalent to $\varphi \leftrightarrow (P \wedge Q)$.

In Alt-Ergo, instead of generating new propositional variables for proxies, we simply *consider* hash-consed (sub)formulas as being their own proxies.

To gain even more efficiency, the proxy (or hash-value now) φ associated to a formula directly represents its pclauses, and it's only when φ or its negation $\neg\varphi$ is asserted by the SAT solver that φ is opened to get back the corresponding pclause. Thus, in the previous example, only the clause $\bar{P} \vee \bar{Q}$ has to be introduced when $\neg\varphi$ is assumed and only P and Q when φ is asserted. The following table shows the pclauses introduced by the solver when it asserts a proxy φ (or its negation) representing formulas made with standard boolean connectives (P is a propositional variable, φ_i are proxies).

Proxy	φ asserted	$\neg\varphi$ asserted
$\varphi \leftrightarrow P$	$\{P\}$	$\{\bar{P}\}$
$\varphi \leftrightarrow \varphi_1 \wedge \varphi_2$	$\{\varphi_1\}$ $\{\varphi_2\}$	$\{\neg\varphi_1 \vee \neg\varphi_2\}$
$\varphi \leftrightarrow \varphi_1 \vee \varphi_2$	$\{\varphi_1 \vee \varphi_2\}$	$\{\neg\varphi_1\}$ $\{\neg\varphi_2\}$
$\varphi \leftrightarrow (\varphi_1 \rightarrow \varphi_2)$	$\{\neg\varphi_1 \vee \varphi_2\}$	$\{\varphi_1\}$ $\{\neg\varphi_2\}$

In order to improve sharing, we have defined an equality relation (and a suitable hashing function) that attempts to identify some subformulas that are logically equivalent. For instance, proxies for $\varphi_1 \rightarrow \varphi_2$ and $\neg\varphi_1 \vee \varphi_2$ would refer to the same variable.

The pclauses handled by the SAT solver implemented in Alt-Ergo have the following forms:

$$\begin{array}{lll} \varphi & := & \texttt{Literal}(l) \\ & | & \texttt{Unit}(\varphi,\varphi) \\ & | & \texttt{Clause}(\varphi,\varphi) \\ & | & \texttt{Forall}(\bar{x},\bar{p},\varphi) \\ & | & \texttt{Exists}(\bar{x},\varphi) \end{array}$$

A pclause is either a single literal $\texttt{Literal}(l)$, a conjunction of two proxies $\texttt{Unit}(\varphi_1,\varphi_2)$, a disjunction of two proxies $\texttt{Clause}(\varphi_1,\varphi_2)$. In addition, a pclause can also represent a quantified formula $\texttt{Forall}(\bar{x},\bar{p},\varphi)$, where $\bar{x}$ denotes the universally quantified variables, $\bar{p}$ the pattern associated to the formula and φ the proxy associated to the body of the quantified formula. Similarly, $\texttt{Exists}(\bar{x},\varphi)$ represents an existentially quantified formula.

This incremental, or *lazy*, CNF conversion technique has been first introduced by Detlef *et. al* in the Simplify theorem prover [26]. The novelty of our approach rests on its combination with the hash-consing mechanism.

Lazy CNF has a strong impact on Alt-Ergo's performances. This is not surprising since, as already noticed by Detlef *et. al* [26],

> "introducing lazy CNF into Simplify avoided such a host of performance problems that [..] it converted a prover that didn't work in one that did."

The main reason is that lazy CNF reduces the number of ground terms sent to the matching module (only the terms contained in the pclause are sent) and, consequently, limits the generation of lemma instantiations.

3.4 The SAT Solver Module

I will present in this section the SAT solver of Alt-Ergo. Some parts of this module are described in [28, 19] and a reflexive Coq tactic is also based on the same ideas [47, 48].

The SAT solver of Alt-Ergo is far from being competitive with the highly efficient solvers engaged in the annual SAT competition. Its main originality is to directly handle formulas in lazy CNF. It also implements a dependency mechanism for producing proof explanations. As a side effect, this mechanism is used for implementing non-chronological backtracking (*a.k.a.* backjumping) [65]. The SAT solver is written in a purely functional style and its proof trees are described by the set of inference rules given in Figures 3.2, 3.3, 3.5, 3.4 and 3.6. Nodes in the proof trees are represented by sequents of the $\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E$, where

– Γ is a set of labeled proxies, written $\varphi[d]$, where φ is a proxy and d a set of proxies, called from now on *explanation*;

– T is a set of labeled literals, written $l[d]$, where l is a literal and d an explanation.
– M is a set of universally quantified formulas associated with explanations, written $(\forall \bar{x}.\varphi)[d]$.
– Δ is a list of unit or binary labeled clauses, denoted (for the sake of readability) $\varphi[d]$ and $(\varphi_1 \vee \varphi_2)[d]$, respectively.
– E is a set of explanations.

Given a sequent $\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E$ in a proof tree, the labeled proxies $\varphi[d]$ in Γ are assumed to be true by the SAT solver (at this precise location of the proof search). Proxies in d are responsible for the introduction of φ in Γ. Similarly, the ground labeled literals $l[d]$ in T are considered as true by the built-in decision procedures, with their associated explanations. Annotated lemmas in M are used by the matching module for generating new (labeled) formulas. The list Δ is the current formula encoded in lazy CNF (it is implicitely viewed as a conjunction of clauses). Each clause in Δ is annotated by a set of proxies that plays a role in its reduction. Finally, E contains the proxies used to establish the proof.

Rules in Figure 3.2 describe the processing of a labeled unit clause $\varphi[d]$ in Δ. Assume-Remove removes the clause if φ is already present in Γ, regardless of its explanation. The following rules moves $\varphi[d]$ from Δ to Γ and, depending on the nature of the proxy φ:

– if φ is a conjunction of unit clauses $\texttt{Unit}(\varphi_1, \varphi_2)$, then Assume-U adds $\varphi_1[d]$ and $\varphi_2[d]$ in Δ;
– if φ is a disjunction of proxies $\texttt{Clause}(\varphi_1, \varphi_2)$, then Assume-C adds the binary clause $(\varphi_1 \vee \varphi_2)[d]$ in Δ;
– if φ is a lemma $\texttt{Forall}(\bar{x}, \bar{p}, \varphi')$, then Assume-Ax adds the labeled universal formula $(\forall \bar{x}[\bar{p}].\varphi')[d]$ in M, where the list $\bar{p}$ is the trigger for this axiom;
– if φ is a ground literal l, then Assume-Lit adds the labeled literal $l[d]$ in T;
– if φ is an existential formula $\texttt{Exists}(\bar{x}, \varphi')$, then Assume-Sko adds the Skolemized formula $\sigma_{\text{sko}}(\varphi')[d]$ in Δ, where σ_{sko} is an appropriate Skolemization substitution.

Rules in Figure 3.3 describe Boolean Constraint Propagation (BCP). BCP-Elim-1 and BCP-Elim-2 eliminate a clause $(\varphi_1 \vee \varphi_2)[d]$ from Δ if φ_1 is already assumed to be true in Γ (Elim-1) or if φ_1 is of the form $\texttt{Literal}(l)$ where l is a ground literal implied by T modulo theory reasoning (Elim-2), written $\mathsf{T} \models l$. BCP-Red-1 and BCP-Red-2 reduce a binary $\varphi_1 \vee \varphi_2[d]$ to a unary clause $\varphi_2[d \cup d']$ if $\neg\varphi_1[d']$ is in Γ (Red-1), or if φ_1 is of the form $\texttt{Literal}(l)$ where l is a ground literal whose negation is implied by T modulo theory reasoning (Elim-2), written $\mathsf{T} \models \neg l : d'$, where d' is the subset of T used by the built-in decision procedures to produce the logical implication.

Rules in Figure 3.4 detect boolean conflicts. These two rules close a branch when Δ contains a proxy (unary clause) $\varphi[d]$ and $\neg\varphi$ is in Γ with some explanation

$$\textsc{Assume-Remove}\ \frac{\Gamma, \varphi[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E}{\Gamma, \varphi[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

$$\textsc{Assume-U}\ \frac{\varphi = \texttt{Unit}(\varphi_1, \varphi_2) \qquad \Gamma, \varphi[d] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi_1[d], \varphi_2[d] : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

$$\textsc{Assume-C}\ \frac{\varphi = \texttt{Clause}(\varphi_1, \varphi_2) \qquad \Gamma, \varphi[d] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, (\varphi_1 \vee \varphi_2)[d] : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

$$\textsc{Assume-Ax}\ \frac{\varphi = \texttt{Forall}(\bar{x}, \bar{p}, \varphi') \qquad \Gamma, \varphi[d] \mid \mathsf{T} \mid \mathsf{M}, (\forall \bar{x}[\bar{p}].\varphi')[d] \vdash \Delta : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

$$\textsc{Assume-Lit}\ \frac{\varphi = \texttt{Literal}(l) \qquad \Gamma, \varphi[d] \mid \mathsf{T}, l[d] \mid \mathsf{M} \vdash \Delta : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

$$\textsc{Assume-Sko}\ \frac{\varphi = \texttt{Exists}(\bar{x}, \varphi') \qquad \Gamma, \varphi[d] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \sigma_{\text{sko}}(\varphi')[d] : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : E}$$

Figure 3.2: SAT: processing of unit clauses

$$\textsc{BCP-Elim-1}\ \frac{\Gamma, \varphi_1[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E}{\Gamma, \varphi_1[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, (\varphi_1 \vee \varphi_2)[d] : E}$$

$$\textsc{BCP-Elim-2}\ \frac{\varphi_1 = \texttt{Literal}(l) \qquad \mathsf{T} \models l \qquad \Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, (\varphi_1 \vee \varphi_2)[d] : E}$$

$$\textsc{BCP-Red-1}\ \frac{\Gamma, \neg\varphi_1[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi_2[d \cup d'] : E}{\Gamma, \neg\varphi_1[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, (\varphi_1 \vee \varphi_2)[d] : E}$$

$$\textsc{BCP-Red-2}\ \frac{\varphi = \texttt{Literal}(l) \qquad \mathsf{T} \models \neg l : d' \qquad \Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi_2[d \cup d'] : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, (\varphi_1 \vee \varphi_2)[d] : E}$$

Figure 3.3: SAT: BCP rules

d' (CONFLICT-1), or if φ is a ground literal $\texttt{Literal}(l)$ and $\neg l$ is implied by T modulo theory reasoning (CONFLICT-2) with d' as an explanation. In both cases, the explanation of the conflict is the union set $d \cup d'$.

$$\textsc{Conflict-1}\ \frac{}{\Gamma, \neg\varphi[d'] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : (d \cup d')}$$

$$\textsc{Conflict-2}\ \frac{\varphi = \texttt{Literal}(l) \qquad \mathsf{T} \models \neg l : d'}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta, \varphi[d] : (d \cup d')}$$

Figure 3.4: SAT: Conflict detection rules

Rules in Figure 3.5 describe the (non-chronological) backtracking mechanism implemented in the SAT solver. Both rules start by picking a proxy φ_1 from a binary clause $(\varphi_1 \vee \varphi_2)[d]$ in Δ and add $\varphi_1[\varphi_1]$ to the context Γ. The explanation $[\varphi_1]$ says that φ_1 is the only reason why it is in Γ (it's a decision made by the solver). BACKJUMPING or BACKTRACKING are then used, depending on whether φ_1 occurs in explanation E or not. If not, then trying the other branch would lead to the same result, so BACKJUMPING concludes that $\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta$ is derivable with the same explanation E. On the contrary, if φ_1 is part of the explanation then BACKTRACKING explores the right branch where φ_1 is false and propagates the explanation E by adding $\neg\varphi_1[E \setminus \varphi_1]$ in Γ.

$$\textsc{BackJumping}\ \frac{(\varphi_1 \vee \varphi_2)[d] \in \Delta \qquad \Gamma, \varphi_1[\varphi_1] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E \qquad \varphi_1 \notin E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E}$$

$$\textsc{BackTracking}\ \frac{\begin{array}{c}(\varphi_1 \vee \varphi_2)[d] \in \Delta \qquad \Gamma, \varphi_1[\varphi_1] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E \\ \Gamma, \neg\varphi_1[E \setminus \varphi_1] \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E' \qquad \varphi_1 \in E\end{array}}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \Delta : E'}$$

Figure 3.5: SAT: (non-chronological) Backtracking

Finally, MROUND in Figure 3.6 calls the matching algorithm to generate instances of lemmas when the set Δ is empty. The matching function $\mathcal{M}$ is described in the next section. The strategy implemented in the SAT solver gives priority to conflict, unit clause and BCP rules (in that order) over (non-chronological) backtracking and matching rules.

$$\textsc{Mround}\ \frac{(\forall \bar{x}[\bar{t}].\varphi)[d] \in \mathsf{M} \qquad \mathcal{M}(\mathsf{T}, \bar{t}) = \sigma \qquad \Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \sigma(\varphi)[d] : E}{\Gamma \mid \mathsf{T} \mid \mathsf{M} \vdash \varnothing : E}$$

Figure 3.6: SAT: Lemma instantiation

3.5 Matching

The matching module is responsible for the generation of ground instances of universal formulas. It implements the function $\mathcal{M}$ used by the SAT solver. $\mathcal{M}$ is based on the algorithm described in [26] extended to account for the instantiation of type variables. The material of the Section is described in [8].

Up to now, I remained vague about the syntax of types, ground terms and patterns. To be more precise, they are defined by the following syntax (for the sake of simplicity, I omit record and enumerative data types):

$$\begin{array}{rcl} \tau & := & \alpha \mid (\tau_1, \ldots, \tau_n)\,\mathtt{t} \\ a & := & x^\tau \mid \mathtt{f}(a_1, \ldots, a_n)^\tau \end{array}$$

A type τ is a type variable α or an abstract type $\mathtt{t}$ with n parameters. Constant types are abstract types with no parameters. A pattern or a term is a variable x or an application of a function symbol $\mathtt{f}$ to n arguments decorated with a type τ (constants are function applications where $n = 0$). Ground terms are just terms with no type and no term variables. A type substitution, denoted by δ, is a set of bindings $\{x_i \mapsto \tau_1\}$ from type variables to types. Similarly, term substitutions, from term variables to term, are denoted by σ. Two term substitutions σ_1 and σ_2 are said *compatible* when $\sigma_1(x) = \sigma_2(x)$ for every x for which there exist bindings in both σ_1 and σ_2 (the same definition holds for type substitutions).

The definition of $\mathcal{M}$ is given below. The function takes two parameters, a set of ground literals T and a pattern p, and returns a set of pairs of terms and type substitutions.

$$\mathcal{M}(\mathsf{T}, x^\tau) = \left\{ (\{x \mapsto a\}, \delta) \mid a^{\delta(\tau)} \in \mathsf{T} \right\}$$

$$\mathcal{M}(\mathsf{T}, \mathtt{f}(p_1, \ldots, p_n)^\tau) = \left\{ (\bigcup \sigma_i, \delta \cup \bigcup \delta_i) \;\middle|\; \begin{array}{l} \mathtt{f}(a_1, \ldots, a_n)^{\delta(\tau)} \in \mathsf{T} \\ \mathsf{T}_i = \{b_i \mid b_i \in \mathsf{T} \wedge \mathsf{T} \models a_i = b_i\} \\ \forall i, \mathcal{M}(\mathsf{T}_i, p_i) \neq \varnothing \\ \forall i, (\sigma_i, \delta_i) \in \mathcal{M}(\mathsf{T}_i, p_i) \\ \quad \sigma_1, \ldots, \sigma_n \text{ compatible} \\ \quad \delta, \delta_1, \ldots, \delta_n \text{ compatible} \end{array} \right\}$$

If p is a term variable x of type τ, then any ground term a in T whose type is an instance $\delta(\tau)$ matches the pattern. If p is of the form $\mathtt{f}(p_1, \ldots, p_n)^\tau$, then $\mathcal{M}$ first looks for all terms $f(a_1, \ldots, a_n)^{\delta(\tau)}$ in T. Recursively, the function is called for matching patterns p_i on the equivalence classes T_i of a_i. These classes are generated by the literals in T modulo the free theory of equality. If none of these calls return an empty set, then the resulting substitutions are checked for compatibility and merged.

For the sake of simplicity, I only considered the case of triggers with a single pattern. It is straightforward to extend the definition for a list of patterns, by considering each pattern in turn and discarding incompatible substitutions.

3.6 The Equality Module

In this section, I describe a decision procedure for combining the theory of equality with uninterpreted function symbols, the theory of AC symbols and an arbitrary theory X equipped with a specific interface described bellow. This module is based on the following three algorithms:

1. CC(X), a congruence closure modulo X. This algorithm has been presented in [16] and its correctness proof has been realized in the Coq proof assistant by Stéphane Lescuyer [46].
2. UF(X), a *union-find* data structure modulo X.
3. AC(X), an extension of AC ground completion modulo X. This algorithm has been presented in [13].

I shall refer to these papers for further explanations and technical details.

Theory Interface. For efficiency reasons, values handled by decision procedures are usually not terms but specific and opaque data structures. In order to present a formalization closed to the implementation, I thus assume that the decision procedure for X handles *semantic values* defined in an abstract domain $\mathcal{R}$. For modularity reasons, the exact structure of semantic values is left unspecified. It is just known that they are somehow constructed of interpreted and uninterpreted (foreign) parts. To compensate, the interface of X provides two functions $[\![\cdot]\!]$ and $\mathcal{L}(\cdot)$ which are reminiscent of the variable abstraction mechanism found in Nelson-Oppen method. The first function constructs a semantic value from out of a term and possibly returns a set of new constraints to satisfy; the second one returns the set of maximal uninterpreted values, called *leaves*, a given semantic value consists of. Additionally, X comes with an equality predicate $\equiv$ on abstract values, a Shostak solver `solve` and an implication relation $\models_X$ for reasoning about built-in predicates (other than the equality symbol).

CC(X). The algorithm is defined by the reduction rules given in Figures 3.7 to 3.11. The rules describe the evolution of the state of the algorithm represented by configurations of the form U | R | UF | L, where

- U is a map from semantic values to set of terms or built-in predicates annotated with explanations which intuitively binds each semantic value with the terms that "use" it.
- R is a set of literals of the form $P(\bar{v})$, where P is a built-in predicate symbol of X and $\bar{v}$ a list of semantic values.
- UF is a union-find data structure on terms modulo X extended with an explanation mechanism. The representative of a term a is the semantic values returned by UF.`find`(a). The union of terms is written UF.`union`(a_1, a_2, d) where d is an explanation for the union. In UF, union of terms means solving equations modulo X. When the union is inconsistent, this operation returns the annotated value $\bot[d]$, where d explains the conflict. Otherwise, it returns a new structure and a substitution between semantic values. Similarly, `distinct` predicates are handled by UF.`distinct`($\bar{a}, d$). The data structure comes also with a function `explain` such that UF.`explain`(a) gives the explanation for t to be equal to its current representative. Finally, UF.`add`(a, d) initializes UF with a new entry a associated to $[\![a]\!]$ and the explanation d and returns the (possibly) new constraints generated by $[\![a]\!]$.
- L is a list of equations and built-in predicates on terms annotated with explanations.

Given an initial list L of annotated (dis)equations and built-in predicates, CC(X) starts in the configuration $\varnothing$ | $\varnothing$ | $\varnothing$ | L.

InitTerm in Figure 3.7 is used when the first literal $l[d]$ in L contains a term $f(\bar{a})$ that has not been yet initialized in UF. Its side condition ensures that all its subterms have been initialized. First, $f(\bar{a})$ is added in UF and a new list of constraints L_1 is generated. Then, U is updated by adding the information that $f(\bar{a})$ uses all the leaves of its direct subterms. Finally, the rule performs congruence closure by looking for equations involving the new term $f(\bar{a})$. This is the construction of the list of equations L_2 where the restrictive condition over $f(\vec{b})$ ensures that only relevant terms are considered.

Rules in Figure 3.8 describe the treatment of built-in and `distinct` literals, assuming InitTerm has been applied. Built-in handles predicates $P(\bar{a})[d]$ where P is a built-in symbol. It first updates U as in InitTerm. Then, a semantic version $P(\bar{v})$ is built and added in R. It also generates new constraints implied by $R \cup P(\bar{v})$ modulo X. Distinct updates UF with distinct terms coming from a predicate `distinct`($\vec{a}$)[d]. It also performs contra-congruence for subterms of the form $f(b_i)$.

Rules in Figure 3.9 handles equations. Remove removes an equation that is already known to be true. The rule Congruence is more complex. If the equation

InitTerm

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (l[d]; \mathsf{L}) \longrightarrow \mathsf{U} \uplus \mathsf{U}' \mid \mathsf{R} \mid \mathsf{UF}' \mid (\mathsf{L}_1; \mathsf{L}_2; l[d]; \mathsf{L})$

if l contains a term $f(\bar{a}) \notin \mathsf{UF}$ and $\forall a_i \in \bar{a},\ a_i \in \mathsf{UF}$
where,

$$\mathsf{UF}', \mathsf{L}_1 = \mathsf{UF}.\mathtt{add}(f(\bar{a}), d)$$
$$\mathsf{U}' = \bigcup_{a_i \in \bar{a}} \bigcup_{v \in \mathcal{L}(\mathsf{UF}.\mathtt{find}(a_i))} v \mapsto \mathsf{U}(v) \cup \{f(\bar{a})\}$$
$$\mathsf{L}_2 = \left\{ f(\bar{a}) = f(\bar{b})[d'] \;\middle|\; \begin{array}{l} \mathsf{UF}.\mathtt{find}(\bar{a}) \equiv \mathsf{UF}.\mathtt{find}(\bar{b}) \\ f(\bar{b}) \in \bigcup_{a_i \in \bar{a}} \bigcap_{v \in \mathcal{L}(\mathsf{UF}.\mathtt{find}(a_i))} \mathsf{U}(v) \\ d' = \bigcup_{a_i \in \bar{a}} \mathsf{UF}.\mathtt{explain}(a_i) \cup \bigcup_{b_i \in \bar{b}} \mathsf{UF}.\mathtt{explain}(b_i) \end{array} \right\}$$

Figure 3.7: CC(X) : Term initialization

$t_1 = t_2[d]$ is not trivial, then UF is updated with a `union` call. The resulting substitution $\{p_i \mapsto v_i\}_{i \in I}$ indicates that every *pivot* p_i has been substituted by v_i in UF'. This substitution is used to update the map U: the terms that used a semantic value p_i before now also use all the values u in the leaves of v_i. A set L_1 of new equations $f(\bar{a}) = f(\bar{b})[d']$ is computed in the following way: $f(\bar{a})$ used a pivot p_i and $f(\bar{b})$ used also p_i or every leaves $u \in \mathcal{L}(r)$ for some semantic values that contained p_i before the union. This rather complicated condition ensures that only relevant terms are considered for congruence. Similarly, a set L_2 of new predicated is calculated.

Finally, rules in Figure 3.10 detect conflicts. Their role consists exclusively in building the most precise explanation when a conflict is detected when processing built-in predicates (Built-in-Conflict), `distinct` predicates (Distinct-Conflict) and equations (Congruence-Conflict).

Case splits. To perform case-split analysis, we suppose that the implication relation $\models_X$ of X is extended for producing coherent choices *w.r.t* a set of literals. Choices fix the values of terms for which a case-split analysis is necessary. Case-split is implemented by the rules in Figure 3.11.

The first three components of CC(X) are duplicated, a configuration is now of the form $(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid \mathsf{L}$, where $(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF})$ is the *main* state and $(\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f)$ the *duplicated* state.

The rule Normal-Run applies the rules in Figures 3.7, 3.8 and 3.9 to both states, as long as no conflict is detected. It a conflict is detected in the main state by Conflict, then the procedure halts. Otherwise, if a choice C can be made,

Built-in

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (P(\bar{a})[d]; \mathsf{L}) \longrightarrow \mathsf{U} \uplus \mathsf{U}' \mid \mathsf{R}, P(\bar{v})[d'] \mid \mathsf{UF} \mid \mathsf{L}'; \mathsf{L})$

where,

$\mathsf{U}' = \bigcup_{a_i \in \bar{a}} \bigcup_{r \in \mathcal{L}(\mathsf{UF}.\mathtt{find}(a_i))} r \mapsto \mathsf{U}(r) \cup \{P(\bar{a})[d]\}$

$d' = d \cup \bigcup_{a_i \in \bar{a}} \mathsf{UF}.\mathtt{explain}(a_i)$

$\bar{v} = \mathsf{UF}.\mathtt{find}(\bar{a})$

$\mathsf{R}, P(\bar{v})[d'] \models_X \mathsf{L}'$

Distinct

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (\mathtt{distinct}(\bar{a})[d]; \mathsf{L}) \longrightarrow \mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid \mathsf{L}'; \mathsf{L}$

where,

$\mathsf{UF}' = \mathsf{UF}.\mathtt{distinct}(\bar{a}, d)$

$\mathsf{L}' = \left\{ \mathtt{distinct}(b_1, \ldots, b_n)[d'] \;\middle|\; \exists a_{k_1} \ldots a_{k_n} \in \bar{a}.\, \mathsf{UF}.\mathtt{find}(a_{k_i}) = f(b_i) \right\}$

Figure 3.8: CC(X): Built-in predicates

Case-Split applies it only to the duplicated state: this is where the two states diverges. If the treatment of C yields to an incoherent duplicated state, then C is negated by Case-Split-Progress. If processing $\neg\mathsf{C}$ still produces a incoherent state, then Case-Split-Conflict halts the procedure. Finally, if the choices made by the theory have become inconsistent with literals in L, then they have to be erased: this is done by Case-Split-Erase-Choices which replaces the duplicated part by a copy of the main state.

$\mathsf{UF(X)}$. The data structure is described in Figures 3.12, 3.13 and 3.14. It implements a union-find data structure for the union of the theories X and AC. The treatment of AC symbols is done by the application of a rewriting system built by ground AC completion modulo X. The state of the data structure is represented by a triplet $\mathsf{EQ} \mid \mathsf{AC} \mid \mathsf{NQ}$, where

- EQ is a map of bindings of the form $a \mapsto u[d]$, where a is a term, u a semantic value called *representative* and d an explanation.
- NQ is a map of binding of the form $a \mapsto \mathcal{S}$, where a is a semantic values and $\mathcal{S}$ is a set of annotated predicates $\mathtt{distinct}(b_1, \ldots, a, \ldots, b_n)[d]$.
- AC is a rewriting system with rewriting rules of the form $\{f(\bar{x}) \mapsto u_i[d]\}$,

Remove

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (a = b[d]; \mathsf{L}) \longrightarrow \mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid \mathsf{L}$

if $\mathsf{UF}.\texttt{find}(a) \equiv \mathsf{UF}.\texttt{find}(b)$

Congruence

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (t_1 = t_2[d]; \mathsf{L}) \longrightarrow \mathsf{U} \uplus \mathsf{U}' \mid \mathsf{R} \mid \mathsf{UF}' \mid (\mathsf{L}_1; \mathsf{L}_2; \mathsf{L})$

if $\mathsf{UF}.\texttt{find}(t_1) \not\equiv \mathsf{UF}.\texttt{find}(t_2)$

where,

$$\mathsf{UF}', \{p_i \mapsto v_i\}_{i \in I} = \mathsf{UF}.\texttt{union}(t_1, t_2, d)$$

$$\mathsf{U}' = \bigcup_{p_i} \bigcup_{u \in \mathcal{L}(v_i)} u \mapsto \mathsf{U}(u) \cup \mathsf{U}(p_i)$$

$$\mathsf{L}_1 = \left\{ f(\bar{a}) = f(\bar{b})[d'] \;\middle|\; \begin{array}{l} f(\bar{a}) \in \mathsf{U}(p_i) \\ f(\bar{b}) \in \mathsf{U}(p_i) \cup \bigcup_{t | p_i \in \mathcal{L}(\mathsf{UF}.\texttt{find}(t))} \bigcap_{u \in \mathcal{L}(\mathsf{UF}'.\texttt{find}(t))} \mathsf{U}(u) \\ d' = \bigcup_{a_i \in \bar{a}} \mathsf{UF}.\texttt{explain}(a_i) \cup \bigcup_{b_i \in \bar{b}} \mathsf{UF}.\texttt{explain}(b_i) \\ \mathsf{UF}'.\texttt{find}(\bar{a}) \equiv \mathsf{UF}'.\texttt{find}(\bar{b}) \end{array} \right\}$$

$$\mathsf{L}_2 = \left\{ P(\bar{a})[d_p] \;\middle|\; P(\bar{a})[d_p] \in \mathsf{U}(p_i) \cup \bigcup_{t | p_i \in \mathcal{L}(\mathsf{UF}.\texttt{find}(t))} \bigcap_{u \in \mathcal{L}(\mathsf{UF}'.\texttt{find}(t))} \mathsf{U}(u) \right\}$$

Figure 3.9: CC(X): Equalities and built-in predicates

where $f(\vec{x})$ is a term and f is an AC symbol, $\bar{x}$ a list of variables, r a semantic value and d an explanation.

For the sake of simplicity, EQ and NQ are used both as a map and a function. In the initial state of the structure the three maps are empty.

Each function defined in Figures 3.12 to 3.14 takes the state of the structure as a first argument. Thus, instead of writing function calls with a dot notation style as for instance in $\mathsf{UF}.\texttt{find}(a)$, we now simply write $\texttt{find}(\mathsf{EQ} \mid \mathsf{AC} \mid \mathsf{NQ}, a)$.

Functions `add`, `find` and `explain` of UF are defined in Figure 3.12. The function `add` takes a term a and, if it's a new term, initializes its entry in the data structure with a binding $a \mapsto v[d]$, where $v[d]$ is the annotated semantic values obtained by normalizing $[\![a]\!]$ *w.r.t.* EQ and AC. A new binding $v \mapsto \varnothing$ is also added in NQ, if v is not already present in NQ. The function also returns new constraints generated by the computation of the semantic value of a. The function `distinct` takes a list $(a_1, \ldots, a_n)$ of terms to be considered distinct with d as explanation. The function first checks that all terms a_i are not equals *w.r.t.* the current state

Built-in-Conflict

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (P(\bar{a})[d]; \mathsf{L}) \longrightarrow \bot[d']$

where $\mathsf{R}, P(\mathsf{UF}.\texttt{find}(\bar{a}))[d \cup \bigcup_{a_i \in \vec{a}} \mathsf{UF}.\texttt{explain}(a_i)] \models_X \bot[d']$

Distinct-Conflict

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (\texttt{distinct}(\vec{a})[d]; \mathsf{L}) \longrightarrow \bot[d']$

where $\mathsf{UF}.\texttt{distinct}(\vec{a}, d) = \bot[d']$

Congruence-Conflict

$\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid (a = b[d]; \mathsf{L}) \longrightarrow \bot[d']$

where $\mathsf{UF}.\texttt{union}(a, b, d) = \bot[d']$

Figure 3.10: CC(X) : Conflict rules

of the structure. If two terms are equal, then `distinct` returns a value $\bot$ with the appropriate explanation. Otherwise, each a_i binding in NQ is updated with a new literal $\texttt{distinct}(a_1, \ldots, a_n)[d]$. Functions `find` and `explain` are straightforward.

The function `union` is defined in Figure 3.13. It takes two terms a_1 and a_2, and an explanation d for the union. If their respective representatives v_1 and v_2 are both binded to the same `distinct` predicate in NQ, then the union is impossible and a value $\bot$ is returned with the appropriate explanation. Otherwise, if a_1 and a_2 are not already equivalent, then $v_1 = v_2$ is solved by calling the solver of X. If the equation is inconsistent, so is the union and the function terminates with the appropriate explanation. Otherwise, the auxiliary function `update` is called to update the current state according to the substitution σ returned by `solve`. If `update` returns a value $\bot[d'']$ then `union` stops its treatment and propagates this value. Otherwise, `update` (see bellow) returns a new state and a list L of new constraints that are recursively processed.

The update of the data structure and the treatment of AC symbols is described in Figure 3.14. The main function `update` takes a substitution σ and an explanation d as arguments. It first checks the update of NQ raises a conflict. If so, then the value $\bot$ is returned with the appropriate explanation. Otherwise, the rewriting system AC is updated by applying σ to the left and right hand side of each rule. This operation produces a new rewriting system AC' and a set of new equations L_1 (see bellow). Then, the σ bindings which imply AC symbols are extracted and superposed with AC' rewriting rules (see bellow). This operation produces a set of new equations L_2. Finally, the maps EQ and NQ are updated by applying σ.

NORMAL-RUN

$$\frac{\begin{array}{c}\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid \mathsf{L}_1 \longrightarrow^* \mathsf{U}' \mid \mathsf{R}' \mid \mathsf{UF}' \mid \varnothing \\ \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \mathsf{L}_1 \longrightarrow^* \mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}_f \mid \varnothing\end{array}}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid (\mathsf{L}_1; \mathsf{L}_2) \longrightarrow (\mathsf{U}' \mid \mathsf{R}' \mid \mathsf{UF}') \mid\mid (\mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}'_f) \mid\mid \mathsf{L}_2}$$

CONFLICT

$$\frac{\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid \mathsf{L} \longrightarrow^* \bot[d]}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid \mathsf{L} \longrightarrow \bot[d]}$$

CASE-SPLIT

$$\frac{\mathsf{R}_f \models_X \mathsf{C} \qquad \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \mathsf{C} \longrightarrow^* \mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}'_f \mid \varnothing}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid \mathsf{L} \longrightarrow (\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}'_f) \mid\mid \mathsf{L}}$$

CASE-SPLIT-PROGRESS

$$\frac{\begin{array}{c}\mathsf{R}_f \models_X \mathsf{C} \\ \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \mathsf{C} \longrightarrow^* \bot[d] \qquad \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \neg\mathsf{C}[d] \longrightarrow^* \mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}_f \mid \varnothing\end{array}}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid \mathsf{L} \longrightarrow (\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}'_f \mid \mathsf{R}'_f \mid \mathsf{UF}'_f) \mid\mid \mathsf{L}}$$

CASE-SPLIT-ERASE-CHOICES

$$\frac{\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF} \mid \mathsf{L}_1 \longrightarrow^* \mathsf{U}' \mid \mathsf{R}' \mid \mathsf{UF}' \mid \varnothing \qquad \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \mathsf{L}_1 \longrightarrow^* \bot[d]}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid (\mathsf{L}_1; \mathsf{L}_2) \longrightarrow (\mathsf{U}' \mid \mathsf{R}' \mid \mathsf{UF}') \mid\mid (\mathsf{U}' \mid \mathsf{R}' \mid \mathsf{UF}') \mid\mid \mathsf{L}_2}$$

CASE-SPLIT-CONFLICT

$$\frac{\mathsf{R}_f \models \mathsf{C} \qquad \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \mathsf{C} \longrightarrow^* \bot[d] \qquad \mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f \mid \neg\mathsf{C}[d] \longrightarrow^* \bot[d']}{(\mathsf{U} \mid \mathsf{R} \mid \mathsf{UF}) \mid\mid (\mathsf{U}_f \mid \mathsf{R}_f \mid \mathsf{UF}_f) \mid\mid \mathsf{L} \longrightarrow \bot[d']}$$

Figure 3.11: CC(X): Case split analysis

$$
\begin{array}{l}
\texttt{add}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC},\ a) = \\
\quad \texttt{if}\ a \in \mathsf{EQ}\ \texttt{then}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC}, \varnothing) \\
\quad \texttt{else} \\
\qquad \texttt{let}\ (u, \mathsf{L}) = [\![a]\!]\ \texttt{in} \\
\qquad \texttt{let}\ v[d] = \texttt{normal}(u, \mathsf{EQ}, \mathsf{AC})\ \texttt{in} \\
\qquad \texttt{let}\ \mathsf{EQ}' = \mathsf{EQ} \cup \{a \mapsto v[d]\}\ \texttt{in} \\
\qquad \texttt{let}\ \mathsf{NQ}' = \left\{ \begin{array}{ll} \mathsf{NQ} \cup \{v \mapsto \varnothing\} & \text{if } v \notin \mathsf{NQ} \\ \mathsf{NQ} & \text{otherwise} \end{array} \right. \quad \texttt{in} \\
\qquad (\mathsf{EQ}' \mid \mathsf{NQ}' \mid \mathsf{AC}, \mathsf{L})
\end{array}
$$

$$
\begin{array}{l}
\texttt{distinct}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC}, (a_1, \ldots, a_n), d) = \\
\quad \texttt{if}\ \exists i, j \in \{1, \ldots, n\}\ \texttt{such that} \\
\qquad \texttt{let}\ u_i[d_i] = \mathsf{EQ}(a_i)\ \texttt{in} \\
\qquad \texttt{let}\ u_j[d_j] = \mathsf{EQ}(a_j)\ \texttt{in} \\
\qquad \texttt{let}\ v_i[d_i'] = \texttt{normal}(u_i, \mathsf{EQ}, \mathsf{AC})\ \texttt{in} \\
\qquad \texttt{let}\ v_j[d_j'] = \texttt{normal}(u_j, \mathsf{EQ}, \mathsf{AC})\ \texttt{in} \\
\qquad v_i \equiv v_j \\
\quad \texttt{then}\ \ \bot[d \cup d_1 \cup d_2 \cup d_1' \cup d_2'] \\
\quad \texttt{else} \\
\qquad \mathsf{NQ} \cup \{a_i \mapsto \texttt{distinct}(a_1, \ldots, a_n)[d] \cup \mathsf{NQ}(a_i)\}_{i \in \{1 \ldots n\}}
\end{array}
$$

$$
\texttt{find}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC},\ a) = \texttt{let}\ v[d] = \mathsf{EQ}(a)\ \texttt{in}\ v
$$

$$
\texttt{explain}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC},\ a) = \texttt{let}\ v[d] = \mathsf{EQ}(a)\ \texttt{in}\ d
$$

Figure 3.12: UF(X) : `add`, `distinct`, `find` and `explain`

```
union (EQ | NQ | AC, a1, a2, d) =
  let u1, d1 = EQ(a1) in
  let u2, d2 = EQ(a2) in
  let v1, d1' = normal(u1, EQ, AC) in
  let v2, d2' = normal(u2, EQ, AC) in
  let d' = d ∪ d1 ∪ d2 ∪ d1' ∪ d2' in
  if ∃distinct(b⃗)[d1''] ∈ NQ(v1) ∧ distinct(b⃗)[d2''] ∈ NQ(v2) then
    ⊥[d' ∪ d1'' ∪ d2'']
  else
    if v1 ≡ v2 then EQ | NQ | AC
    else
      let σ = solve(v1, v2) in
      if σ = ⊥ then ⊥[d']
      else
        let r = update(EQ | NQ | AC, σ, d') in
        match r with
         | ⊥[d''] ⟶ r
         | (EQ' | NQ' | AC', L) ⟶
             fold_list union (EQ' | NQ' | AC') L
```

Figure 3.13: UF(X) : `union`

The auxiliary function `compose_collapse` applies σ and normalizes the left and right parts of rules in AC. The rules thus obtained are kept as rewriting rules as long as their left parts are unchanged. If the left part of a rule is impacted by σ, then it is changed into an equation (that has to be processed by the algorithm). Finally, `head_cp` computes the critical pairs between a substitution σ (viewed as a rewriting system) and the AC rewriting rules.

3.7 The Arithmetic Module

This module implements the theory of arithmetic for integers and real numbers. Its interface is described in Section 3.6.

Semantic values. Semantic values are linear polynomials of the form $\Sigma_i c_i \times x_i + c_0$, where the c_is ($i \in \{0, \ldots, n\}$) are constant numbers, and the indeterminates x_is

$\mathtt{update}\ (\mathsf{EQ} \mid \mathsf{NQ} \mid \mathsf{AC},\ \sigma, d) =$
$\quad \mathtt{if}\ \exists u.\, \mathtt{distinct}(\bar{b})[d_1] \in \mathsf{NQ}(u) \wedge \mathtt{distinct}(\bar{b})[d_2] \in \mathsf{NQ}(u\sigma)\ \mathtt{then}$
$\quad\quad \bot[d \cup d_1 \cup d_2]$
$\quad \mathtt{else}$
$\quad\quad \mathtt{let}\ (\mathsf{AC}', \mathsf{L}_1) = \mathtt{compose_collapse}\ (\mathsf{AC}, \mathsf{EQ}, \sigma, d)\ \mathtt{in}$
$\quad\quad \mathtt{let}\ \sigma_{AC} = \{f(\bar{x}) \mapsto u \mid f(\vec{x}) \mapsto u \in \sigma \text{ and } f \text{ is an AC symbol}\}\ \mathtt{in}$
$\quad\quad \mathtt{let}\ \mathsf{L}_2 = \mathtt{head_cp}\ (\mathsf{AC}', \sigma_{AC}, d)\ \mathtt{in}$
$\quad\quad \mathtt{let}\ \mathsf{EQ}' = \{a \mapsto u\sigma \mid a \mapsto u \in \mathsf{EQ}\}\ \mathtt{in}$
$\quad\quad \mathtt{let}\ \mathsf{NQ}' = \{r\sigma \mapsto \mathsf{NQ}(r) \cup \mathsf{NQ}(r\sigma)\} \cup \mathsf{NQ}\ \mathtt{in}$
$\quad\quad (\mathsf{EQ}' \mid \mathsf{NQ}' \mid \mathsf{AC}' \cup \sigma_{AC}, \mathsf{L}_1 \cup \mathsf{L}_2)$

$\mathtt{compose_collapse}\ (\mathsf{AC}, \mathsf{EQ}, \sigma, d) =$

$$\mathtt{let}\ \mathsf{AC}' = \left\{ l \mapsto r'[d \cup d_l \cup d_{r'}] \;\middle|\; \begin{array}{l} l \mapsto r[d_l] \in \mathsf{AC} \\ l', d_{l'} = \mathtt{normal}(l\sigma, \mathsf{EQ}, \mathsf{AC}) \\ r', d_{r'} = \mathtt{normal}(r\sigma, \mathsf{EQ}, \mathsf{AC}) \\ l = l' \end{array} \right\} \quad \mathtt{in}$$

$$\mathtt{let}\ \mathsf{L} = \left\{ l' = r'[d \cup d_l \cup d_{l'} \cup d_{r'}] \;\middle|\; \begin{array}{l} l \mapsto r[d_l] \in \mathsf{AC} \\ l', d_{l'} = \mathtt{normal}(l\sigma, \mathsf{EQ}, \mathsf{AC}) \\ r', d_{r'} = \mathtt{normal}(r\sigma, \mathsf{EQ}, \mathsf{AC}) \\ l \neq l' \end{array} \right\} \quad \mathtt{in}$$

$\quad \mathsf{AC}', \mathsf{L}$

$\mathtt{head_cp}\ (\mathsf{AC}, \sigma, d) =$

$$\left\{\ u(\beta, t_2) = u(\gamma, t_1)[d \cup d'] \mid u(\vec{x}, \beta) \mapsto t_1 \in \mathsf{AC}[d'] \wedge u(\vec{x}, \gamma) \mapsto t_2 \in \sigma\ \right\}$$

Figure 3.14: UF(X) : ground AC completion modulo X

are terms whose root symbols are (considered as) uninterpreted. First parts of polynomials are implemented by maps from (uninterpreted) terms to constants. For simplicity, we assume maps have the same domain. Second parts are just constants. The function $[\![.]\!]$ is defined in Figure 3.15. For the sake of readability, polynomials are represented in a mathematical way and arithmetic symbols are denoted `plus`, `minus`, `mult`, `div`, and `mod`. Indeterminates are surrounded by quotes and are considered as uninterpreted.

Semantic values are built by induction over the term structure. The first four

cases are immediate. The semantic value for $\texttt{mult}(a_1, a_2)$ is done by multiplying the polynomials returned by $[\![a_1]\!]$ and $[\![a_2]\!]$. Non-linear monomials $a \times x_i x_j$ are represented by linear bindings "$\texttt{mult}(x_i, x_j)$" $\mapsto a$, where symbol mult in indeterminate "$\texttt{mult}(x_i, x_j)$" is uninterpreted. Similarly, the semantic value for $\texttt{div}(a_1, a_2)$ is left uninterpreted when $[\![a_2]\!]$ has at least one (non null) indeterminate or when it is reduced to 0 (as mentioned in Section 3.2, the division operation is a total function). Otherwise, we divide $[\![a_1]\!]$ by (the constant) $[\![a_2]\!]$. If a_1 is of type real, we simply return the polynomial thus obtained. If the division is Euclidean and if a_1 is a constant, then we return the real quotient $[\![a_1]\!]/[\![a_2]\!]$ rounded towards zero. Otherwise, we calculate the polynomial associated with the rest of the division (see bellow) and we return $([\![a_1]\!] - [\![\texttt{mod}(a_1, a_2)]\!])/[\![a_2]\!]$. The polynomial associated with $\texttt{mod}(a_1, a_2)$ is $1 \times$ "$\texttt{mod}(a_1, a_2)$" $+ 0$, unless both $[\![a_1]\!]$ and $[\![a_2]\!]$ are constants; in that case we compute the exact modulo operation. If only $[\![a_2]\!]$ is a constant, we generate the following extra constraints:

$$0 \leq \text{"}\texttt{mod}(a_1, a_2)\text{"} < [\![t_2]\!]$$
$$k \times t_2 + \text{"}\texttt{mod}(a_1, a_2)\text{"} = t_1$$

where k is a *fresh* constant name.

The definition of function $\mathcal{L}(.)$ is straightforward, it simply returns the set of terms x_i not binded in the map to 0.

Solving equations For solving integer equations, we reuse the Omega Test solver defined in [42] (Chapter 5) to turn integer linear equalities into substitutions. Real equations are simply solved using Gaussian elimination.

Built-in predicates. The implication relation $\models_X$ is implemented by a combination of Fourier-Motzkin algorithm and interval-based arithmetic.

Fourier-Motzkin determines whether the polyhedral set associated with a conjunctions of linear constraints over real (or integer) variables is empty [32, 42]. The idea behind the algorithm is to perform successive variable eliminations by projecting its constraints onto the rest of the system, thus generating additional (but simpler) constraints. This procedure can be easily extended to discover new equalities: all inequations involved in the generation of constraints of the form $c \leq b$ where $c = b$, are in fact equalities [45].

The Fourier-Motzkin algorithm is combined with an interval-based arithmetic module which maintains a table of disjoint unions of intervals for every variable – as well as for polynomials – encountered in the original formula. Intervals are refined by the use of three mechanisms.

- For each generated constraint of the form $ax + b \leq 0$, we derive an upper or lower bound for x that are used to reduce the intervals of x. Fourier-Motzkin is also directly exploited to refine the lower bounds of polynomials composing

the inequations used during the elimination process. For example if a set of inequations $\{P_1(\bar{x}) \leq 0, ..., P_n(\bar{x}) \leq 0\}$ leads to the trivial inequation $c \leq 0$ and if we assume that $k_{i,j}$ is the coefficient used in a projection at step j on inequation i then we have $c \leq (\prod_{j=1}^{steps} k_{i;j})P_i(\bar{x}) \leq 0$ for all original inequations that participated in reaching $c \leq 0$. These techniques are generalized to non-linear terms which also have their own entry in the intervals table.
- We perform multiplications and other non-linear arithmetic operations [38] on (union of) intervals.
- We compute implied bounds for elements which compose non-linear terms, e.g. what are the bounds of x, knowing those of x^n?

Theory-based case analysis. Intervals contained in the tables can be used to perform case analysis. We only perform such case analysis for variables or polynomials with finite domains.

3.8 Perspectives

The decision procedures at the heart of Alt-Ergo are mainly based on rewriting techniques: they either provide a canonizer and a solver for, respectively, computing canonical forms of terms and solving equations (e.g. arithmetic, records), or simply behave as a specific rewriting system (e.g arrays). We plan to extend Alt-Ergo with user-defined rewriting rules. For that, we want to study the design a new combination algorithm which will extend the parameterized algorithms AC(X) to handle, in a *uniform* way, theories providing canonizers and solvers, equational axioms (e.g. AC symbols), convergent or simply ground rewriting systems.

$[\![\texttt{n}]\!] = n, \varnothing \qquad \text{if } n \in \mathbb{Z} \cup \mathbb{R}$

$[\![\texttt{f}(a_1, \ldots, a_n)]\!] = 1 \times \text{"}\texttt{f}(a_1, \ldots, a_n)\text{"} + 0, \varnothing \quad \text{if } \texttt{f} \text{ is uninterpreted}$

$[\![\texttt{plus}(a_1, a_2)]\!] =$
  $\texttt{let } (\Sigma_i c_i \times x_i + c_0), \mathsf{L}_1 = [\![a_1]\!] \texttt{ in}$
  $\texttt{let } (\Sigma_i d_i \times x_i + d_0), \mathsf{L}_2 = [\![a_2]\!] \texttt{ in}$
  $\Sigma_i (c_i + d_i) \times x_i + (c_0 + d_0),\ \mathsf{L}_1 \cup \mathsf{L}_2$

$[\![\texttt{minus}(a_1, a_2)]\!] =$
  $\texttt{let } (\Sigma_i c_i \times x_i + c_0), \mathsf{L}_1 = [\![a_1]\!] \texttt{ in}$
  $\texttt{let } (\Sigma_i d_i \times x_i + d_0), \mathsf{L}_2 = [\![a_2]\!] \texttt{ in}$
  $\Sigma_i (c_i - d_i) \times x_i + (c_0 - d_0),\ \mathsf{L}_1 \cup \mathsf{L}_2$

$[\![\texttt{mult}(a_1, a_2)]\!] =$
  $\texttt{let } (\Sigma_i c_i \times x_i + c_0), \mathsf{L}_1 = [\![a_1]\!] \texttt{ in}$
  $\texttt{let } (\Sigma_i d_i \times x_i + d_0), \mathsf{L}_2 = [\![a_2]\!] \texttt{ in}$
  $(\Sigma_i \Sigma_j c_i d_j \times \text{"}\texttt{mult}(x_i, x_j)\text{"}) + (\Sigma_i (c_i d_0 + d_i c_0) \times x_i) + c_0 d_0,\ \mathsf{L}_1 \cup \mathsf{L}_2$

$[\![\texttt{div}(a_1, a_2)]\!] =$
  $\texttt{let } (\Sigma_i c_i \times x_i + c_0), \mathsf{L}_1 = [\![a_1]\!] \texttt{ in}$
  $\texttt{let } (\Sigma_i d_i \times x_i + d_0), \mathsf{L}_2 = [\![a_2]\!] \texttt{ in}$
  $\texttt{if } \exists d_i \neq 0 \text{ or } d_0 = 0 \texttt{ then } 1 \times \text{"}\texttt{div}(a_1, a_2)\text{"} + 0, \mathsf{L}_1 \cup \mathsf{L}_2$
  $\texttt{else}$
    $\texttt{let } r = \Sigma_i \frac{c_i}{d_0} \times x_i + \frac{c_0}{d_0} \texttt{ in}$
    $\texttt{if } r \in \mathbb{R} \texttt{ then } r, \mathsf{L}_1 \cup \mathsf{L}_2$
    $\texttt{else if } \forall i. c_i = 0 \texttt{ then } \lfloor r \rfloor, \mathsf{L}_1 \cup \mathsf{L}_2$
    $\texttt{else}$
      $\texttt{let } \Sigma_k e_k \times x_k + e_0, \mathsf{L} = [\![\texttt{mod}(a_1, a_2)]\!] \texttt{ in}$
      $r - (\Sigma_k \frac{e_k}{d_0} \times x_k + \frac{e_0}{d_0}), \mathsf{L}_1 \cup \mathsf{L}_2 \cup \mathsf{L}$

$[\![\texttt{mod}(a_1, a_2)]\!] =$
  $\texttt{let } (\Sigma_i c_i \times x_i + c_0), \mathsf{L}_1 = [\![a_1]\!] \texttt{ in}$
  $\texttt{let } (\Sigma_i d_i \times x_i + d_0), \mathsf{L}_2 = [\![a_2]\!] \texttt{ in}$
  $\texttt{if } \exists d_j \neq 0 \text{ or } d_0 = 0 \texttt{ then } 1 \times \text{"}\texttt{mod}(a_1, a_2)\text{"} + 0, \mathsf{L}_1 \cup \mathsf{L}_2$
  $\texttt{else if } \forall i. c_i = 0 \texttt{ then } c_0 \% d_0, \mathsf{L}_1 \cup \mathsf{L}_2$
  $\texttt{else}$
    $\texttt{let } l_1 = (\texttt{mod}(a_1, a_2)\text{"} \texttt{>=} 0) \texttt{ in}$
    $\texttt{let } l_2 = (\text{"}\texttt{mod}(a_1, a_2)\text{"} \texttt{<} d) \texttt{ in}$
    $\texttt{let } k = \texttt{fresh_name}() \texttt{ in}$
    $\texttt{let } l_3 = (\texttt{plus}(\texttt{mult}(k, a_2), \text{"}\texttt{mod}(a_1, a_2)\text{"}) = a_1) \texttt{ in}$
    $1 \times \text{"}\texttt{mod}(a_1, a_2)\text{"} + 0, \mathsf{L}_1 \cup \mathsf{L}_2 \cup \{l_1, l_2, l_3\}$

Figure 3.15: Arith : Building semantic values

4
The Cubicle SMT-based Model Checker

This chapter presents Cubicle, an open source SMT-based model checker under the Apache License version 2. Cubicle is available at `http://cubicle.lri.fr`.

4.1 Motivations

I have recently started a collaboration with the Intel Strategic Cad Labs about the design of an SMT-based model checker. With Amit Goel and Sava Krstic, we are developing Cubicle, a new model checker for verifying safety properties of parameterized systems. Cubicle is used to verify safety properties of *array-based systems*. This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [35]. Cache coherence protocols and mutual exclusion algorithms are typical examples of such systems. Cubicle model-checks by a parallel symbolic backward reachability analysis on infinite sets of states using Satisfiabilty Modulo Theories. For that purpose, it is tightly coupled with a lightweight and enhanced version of Alt-Ergo. The main results on this topics are presented in more details in [18].

4.2 Cubicle's Input Language

Cubicle's input language is a typed version of Murφ [27] similar to the one of UCLID [9], rudimentary at the moment, but sufficiently expressive for typical parameterized systems.

A system described in Cubicle starts by a set of type, variable, and array declarations. Cubicle provides four built-in types `int`, `real`, `bool` and `proc`, the abstract data type of processes. Arrays are restricted to be only indexed by `proc` variables. Additionally, the user can specify its own abstract types and enumerations data types.

```
type st = Idle | Want | Crit
```

```
type data

var Timer : real
array State[proc] : st
array Chan[proc] : data
array Flag[proc] : bool
```

The initial states of the system are then defined by a universal conjunction of literals characterizing the values for some variables and array entries. For simplicity, initial formulas are written with the following notation, where `z` denotes a variable of type `proc` implicitly universally quantified:

```
init (z) { Flag[z] = False && State[z] = Idle  && Timer = 0.0 }
```

Safety properties to be verified are expressed in their negated form as formulas that represent unsafe states. Each unsafe formula must be a *cube*, i.e., have the form $\exists \bar{x}.(distinct(\bar{x}) \wedge C)$, where $\bar{x}$ is a set of process variables, $distinct(\bar{x})$ is the conjunction of all disequations between the variables in $\bar{x}$, and C is a conjunction of literals. In the code, we leave the existential quantifier implicit, as well as the *distinct* part, as in:

```
unsafe (x y) { State[x] = Crit && State[y] = Crit }
```

The rest of the system is given by set of guard/action transitions. Each transition may be parameterized by (one or more) process identifiers as in the following example

```
transition t (i j)
requires { i < j && State[i] = Idle && Flag[i] = False &&
           forall_other k. (Flag[k] = Flag[j] || State[k] <> Want) }
{
  Timer := Timer + 1.0;
  Flag[i] := True;
  State[k] := case
              | k = i : Want
              | State[k] = Crit && k < i : Idle
              | _ : State[k]
}
```

where the arguments `i` and `j` are implicitly existentially quantified representing distinct processes. Guards are conjunctions of literals (equations, disequations or inequations) and universal requirements of the form $\forall k. C_1 \vee \ldots \vee C_n$ where k is a process variable implicitly distinct from every parameter of the transition and $C_1 \ldots C_n$

are conjunctions of literals. Actions are array updates and variables assignments. Array updates are coded either as simple assignments like `Flag[i] := True`, or by a case construct like `State[k] := case ...` where `k` is an implicitly universally quantified variable and each condition is a conjunction of literals, and the default case is denoted by `_`.

The transitions define the system's execution: an infinite loop that at each iteration: (1) non-deterministically chooses a transition instance whose guard is true in the current state; and (2) updates state variables according to the action of the fired transition instance. A system is safe if unsafe states are not reachable from an initial state.

4.3 Engineering an Efficient Reachability Modulo Theory Algorithm

Cubicle implements a backward reachability algorithm to check safety properties of array-based systems. Given an initial formula $\mathcal{I}$, a set of transitions $\mathcal{T}$, and an unsafe formula $\mathcal{U}$, this algorithm is described by the following pseudo-code:

1: $V \leftarrow \varnothing$
2: push_queue($Q, \mathcal{U}$)
3: while not_empty(Q)
4: $\varphi \leftarrow$ pop_queue(Q)
5: if $\neg(\varphi \wedge \mathcal{I} \vdash \bot)$ then return(unsafe)
6: if $\neg(\varphi \vdash \bigvee_{\psi \in V} \psi)$ then
7: $V \leftarrow V \cup \{\varphi\}$
8: push_queue(Q, $\mathtt{pre}_{\mathcal{T}}(\varphi)$)
9: return(safe)

The algorithm maintains a set V and a priority queue Q of *visited* and *unvisited cubes* respectively. Initially, V is empty and Q contains the system's unsafe condition. Then, at each iteration, the highest-priority cube φ is taken from Q and is checked for consistency with the initial condition (line 5). If this safety check fails, the algorithm terminates with "system unsafe". If the safety check passes, we proceed to the *subsumption check* $\varphi \vdash \bigvee_{\psi \in V} \psi$ (line 6). If this fails, then φ is added to V and we compute the formula $\mathtt{pre}_{\mathcal{T}}(\varphi)$ (line 8) which represents the set of states from which the states described by φ can be reached in one transition. In array-based systems, if φ is a cube then $\mathtt{pre}_{\mathcal{T}}(\varphi)$ can be also represented as a union (disjunction) of cubes. These cubes are then added to Q, and we move on

to the next iteration. If the subsumption check succeeds, then φ is dropped from consideration and we move on. The algorithm terminates when a safety check fails or Q becomes empty.

Safety checks, being ground satisfiablity queries, are easy for SMT solvers. The challenge is in subsumption checks $\varphi \vdash \bigvee_{\psi \in V} \psi$ because of their size and the "existential implies existential" logical form. Assuming $\varphi \triangleq \exists \bar{x}.F$ and $\psi \triangleq \exists \bar{y}.G_\psi$ ($\psi \in V$), the subsumption check translates into the validity check for the ground formula

$$H \triangleq (F \Rightarrow \bigvee_{\psi \in V} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma)$$

where Σ is the set of all substitutions from $\bar{y}$ to $\bar{x}$.

Assuming $|V| = 20000$ and $|\Sigma| = 120$ (*e.g.* if $|\bar{x}| = |\bar{y}| = 5$), then the number of ground clauses to consider is approximately

$$|\bigvee_{\psi \in \mathbf{V}} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma| \sim 2.10^6$$

Now, viewing any cube $G_\psi\sigma$ as a set of literals, one can make two useful comparisons with F:

1. if $G_\psi\sigma$ is a subset of F, then H is valid;
2. if $G_\psi\sigma$ contains a literal that directly contradicts a literal of F, then $G_\psi\sigma$ is redundant.

Cubicle aggressively attempts to prove H by building and verifying it incrementally, adding one disjunct to its consequent at a time. Essentially, it examines all pairs (ψ, σ) one-by-one, stopping the process when the current overapproximation of H becomes known to be valid. For each pair (ψ, σ), the cube $G_\psi\sigma$ is first checked for redundancy; if redundant, it is ignored and a new pair (ψ, σ) is processed. If not redundant, the cube is subject to the subset check for $F \vdash G_\psi\sigma$. If this check succeeds, H is claimed valid; otherwise $G_\psi\sigma$ gets added to H (as a disjunct of its consequent) and the SMT solver checks if the newly obtained (weakened) H becomes valid.

Cubicle's integration with the SMT solver at the API level is crucial for efficient treatment of the subsumption check. For any such check, a single context for the SMT solver is used; it just gets incremented and repeatedly verified. To support the efficient (symmetry-reduced) and exhaustive application of the inexpensive redundancy and subset checks, cubes are maintained in normal form where variables are renamed and implied literals removed at construction time.

The strategy for exploring the search space is also essential. It pays to visit as few cubes as possible, which suggess giving priority to more "generic" cubes (those that represent larger sets of states). Thus, neither breadth-first nor depth-first

search are good in their pure form. By default, Cubicle uses BFS (changeable with the `-search` option to DFS or some variants) combined with a heuristically delayed treatment of some cubes. Currently, a cube is delayed if it introduces new process variables or does not contribute new information on arrays. Finally, Cubicle can remove cubes from V when they become subsumed by a new cube.

Invariant generation. Cubicle supports user-supplied invariants and invariant synthesis, both of which can significantly reduce the search. Subsets of visited nodes that only contain predicates over a unique process variable are used as candidate invariants. Each of them is verified by starting a new resource limited backward reachability analysis. Cubicle can also discover "subtyping invariants" (saying that a variable can take only a selected subset of values) by a static analysis and these invariants can be natively exploited by the SMT solver which supports definitions of subtypes for enumerated data-types.

Multi-Core Architecture A natural way to scale up model checkers is to parallelize their CPU intensive tasks to take advantage of the widespread availability of multi-core machines or clusters [37, 2, 49]. In our framework, this is achieved by parallelizing the backward reachability loop and the generation of invariants. As mentioned above, since invariant synthesis is done independently from the main loop, it is straightforward to do it in parallel. However, concerning the loop itself, a naive parallel implementation would lose the precise guidance of the exploration [1], and more importantly, could break the correctness of the tool because of an unsafe use of some optimizations described in the previous section.

In our setting, the most resource consuming tasks are fixpoints checks which can be hard problems even for efficient SMT solvers. To gain efficiency, we implemented a concurrent version of BFS based on the observation that all such computations arising at the same level of the search tree can be executed in parallel. Our implementation is based on a centralized master/workers architecture. The master assigns fixpoints to workers and a synchronization barrier is placed at each level of the tree to retain a BFS order. The master asynchronously computes the preimages of nodes that are not verified as fixpoints by the workers. In the meanwhile, the master can also assign invariant generation tasks that will be processed by available workers. Finally, to safely delete nodes from $\mathcal{V}$, the master must discard the results about nodes that have been deleted while they were being checked by a worker.

Cubicle provides a concurrent breadth-first exploration of the search space using n parallel processes on a multi-core machine with the `-j` n option. The implementation is based on Functory [29], an OCaml library with a rich functional interface

1. Our experiments showed that a non-deterministic parallel exploration can be worse than a guided sequential search.

which facilitates the execution of parallel algorithms. Functory supports multi-core architectures and distributed networks; it has also a robust fault-tolerance mechanism. Concerning a distributed implementation, one of the main issues is to limit the size of data involved in transactions between the master and the workers. For instance, the size of $\mathcal{V}$ can quickly become a bottleneck in an architecture based on message passing communications. As future work, we plan to develop a distributed implementation that will only need to send updates of data-structures.

4.4 Perspectives

As future work, we would like to harness the full power of the SMT solver by sharing its data structures and even more tightly integrating its features in the model checker. In particular, this would be very useful to discover symmetries and to simplify nodes by finding semantic redundancies modulo theories. We are also interested in exploiting the unsat cores returned by the solver to improve our node deletion mechanism. Finally, we are investigating a new method for generating invariants based on the invisible invariant technique [58].

5
Conclusion and Perspectives

Alt-Ergo is used as one of the back-end provers of the Why platform [30]. It is also used directly by Airbus France as a prover for the Caveat system and intended to be used in production. It is also distributed by Altran-Praxis company as an alternative prover for their Spark/Ada [10] verification tool.

My research perspectives for the Alt-Ergo SMT solver and the Cubicle model checker are directly related to three research projects that have started in September 2012. The development of Alt-Ergo and Cubicle is also part of the new INRIA project Toccata leading by Claude Marché.

The ANR project Bware. The project BWare aims at the design of a new platform for verifying proof obligations coming from the Atelier B system. It aims at improving the level of automation and also the level of trust we can have in the proofs.

A key challenge for Alt-Ergo in this project is to handle efficiently the set of axioms that define the operators of the B method. Some of these axioms are just purely administrative : they represent properties of basic set theoretic operations (AC properties, transitive closures, etc.). Contrary to other lemmas, a huge number of instances of administrative lemmas can be necessary to prove a goal. Handling efficiently such axioms is thus crucial to increase the level of automation. In practice, it turns out that such lemmas can be simply expressed as (conditional) rewriting rules. Therefore, in order to improve Alt-Ergo, we want to study how to integrate such rewriting rules directly in the core of the prover and use them to canonize terms modulo the other built-in theories.

We also want to study how the user can interact with Alt-Ergo to finish a proof that was not automatically proved. The first part of this study will consist in providing information to the user to understand why the theorem prover failed to prove its goal. This information could be provided in real-time as a set of evolving indicators showing the progress of the proof: external indicators (lemma instantiations, case-splits, etc.) and internal indicators (decision procedures used, memory

consumption etc.). In the second part, we will study how the user can interactively guide Alt-Ergo to terminate the proof (manual lemma instantiations, lemma priority, case-splits, selection hypothesis etc.). Finally, we will study how to save these interactions in a file so that Alt-Ergo can replay them, even in the case where the proof context has evolved and is not identical.

Finally, Alt-Ergo makes heavy use of persistent data structures which are expensive in term of memory allocation. In order to improve its performances, we also plan to study how to continue our work on semi-persistent data structures [17], a kind of data structures that exhibit a persistent behavior only when used in a backtracking context.

The ANR project Cafein. The project Cafein "Combinaison d'approches formelles pour l'étude d'invariants numériques" aims at the verification of safety-critical reactive systems using a combination of formal approaches: SMT solving techniques, abstraction interpretation, extended model-checking (bounded model-checking, k-induction).

In this project, we plan to extend Alt-Ergo with a model generation mechanism and to support Craig interpolants. Model generation is particularly challenging in the presence of user-given rewrite rules, since naive algorithms will most likely generate models that do not satisfy all of the rewrite equations. We thus plan to study how to implement a model generator driven by a custom model generator provided by the user for their rewrite rules.

Intel collaboration. We plan to develop, in collaboration with SCL's Goel and Krstíc, a new version of Cubicle with significantly improved model-checking power. This will require innovative algorithmic enhancements: to be found, implemented, and evaluated. Algorithmic investigation will follow at least the following two directions: (1) Simplify Cubicle's main state object (partial proof graph) by on-the-fly applications of "subsumption", "weakening", and possibly other transformations based on the logical contents encoded in the proof graph. (2) Replace the brute-force backward reachability with a more sophisticated search algorithm. Specifically, exlore the localized back-and-forth method ("Rushby graph") introduced by Goel-Krstíc, Majumdar and Tetali.

If the efficiency of Cubicle becomes significantly improved, we would try to use it to verify systems currently out-of-reach: either some complex ("industrial strength") cache coherence protocol, or a complex array-manipulating C-program.

The Toccata project. One of the goals of the Toccata project is to continue the efforts around Alt-Ergo and Cubicle towards more expressivity, efficiency, and usability, in the context of program verification.

In particular, an important challenge is to integrate in Alt-Ergo a decision procedure like the one implemented in the tool Gappa [24]. With Cody Roux and Guillaume Melquiond, we have already started the development of such an extension. The dedicated procedure is based on the approach of Gappa: it performs saturation of consequences of the axioms, in order to refine bounds on expressions. In addition to the original approach, bounds are further refined by a constraint solver for linear arithmetic. Combined with the natural support for equalities provided by the arithmetic module of Alt-Ergo, our approach improves the treatment of goals coming from deductive verification of numeric programs. The first experiments are promising, we need to improve performance of the procedure on large problems, allowing us to treat a wider range of examples from industrial sofware verification. Moreover we have not integrated all the theorems from Gappa. In particular, we only treat absolute error and not yet relative error.

Another perspective is to continue the former work on the certified core of Alt-Ergo [46]. The Ergo Coq tactic should be extended to support more features: more theories (full integer arithmetic, real arithmetic, arrays, etc.), quantifiers. With Denis Cousineau, we have started a promising trace-based approach for Coq that only uses existing tactics of this proof assistant and require a light interaction with Alt-Ergo.

Bibliography

[1] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

[2] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7, 2010.

[3] C. Barrett. *Checking Validity of Quantifier-free formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2002.

[4] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1996.

[5] C. Barrett, D. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FROCOS)*, volume 2309 of *LNAI*, pages 132–147, Santa Margherita Ligure, Italy, 2002. Springer.

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[7] N. Bjørner et al. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 1996.

[8] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability*

Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, pages 1–5, New York, NY, USA, 2008. ACM.

[9] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.

[10] Bernard Carré and Jonathan Garnsworthy. Spark, an annotated ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA '90*, TRI-Ada '90, pages 392–402, New York, NY, USA, 1990. ACM.

[11] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *IJCAR 2012 : Proceedings of the 6th International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, Manchester, UK, June 2012. Springer Verlag.

[12] Sylvain Conchon, Evelyne Contejean, and Mohame Iguernelala. Ground associative and commutative completion modulo shostak theories. In *LPAR 17*, Yogyakarta, Indonesia, 2010. Easychair Proceedings. Short paper.

[13] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized Rewriting and Ground AC Completion Modulo Shostak Theories. In Parosh A. Abdulla and K. Rustan M. Leino, editors, *TACAS 2011: Proceedings of the 17h Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Saarbrucken, Germany, March 2011. Springer Verlag.

[14] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized Rewriting and Ground AC Completion Modulo Shostak Theories: Design and Implementation. *Logical Methods in Computer Science*, To appear.

[15] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In *5th International Workshop on Satisfiability Modulo*, Berlin, Germany, July 2007.

[16] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Cc(x): Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51–69, May 2008.

[17] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-Persistent Data Structures. Research Report 1474, LRI, Université Paris Sud, September 2007.

[18] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In

Madhusudan Parthasarathy and Sanjit A. Seshia, editors, *CAV 2012: Proceedings of the 24th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berkeley, California, USA, July 2012. Springer Verlag.

[19] Sylvain Conchon, Johannes Kanig, and Stéphane Lescuyer. SAT-Micro : petit mais costaud! In *Dix-neuvièmes Journées Francophones des Langages Applicatifs*, Étretat, France, 2008. INRIA.

[20] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. In *Proceedings of the 9th Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 537–553, Warsaw, Poland, April 2003. Springer Verlag.

[21] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354(2):187–210, 2006.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[23] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[24] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Comput.*, 60(2):242–253, February 2011.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, March 2008. Springer.

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

[27] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[28] Jean-Christophe Filliâtre and Sylvain Conchon. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[29] Jean-Christophe Filliâtre and K. Kalyanasundaram. Functory: A distributed computing library for Objective Caml. In *TFP*, pages 65–81, 2011.

[30] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[31] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102, pages 246–249, 2001.

[32] J-B J Fourier. Reported in: Analyse des travaux de l'Académie Royale des Sciences, pendant l'année 1824, Partie mathématique, Histoire de l'Académie Royale des Sciences de l'Institut de France. (7), 1827.

[33] H. Ganzinger. Shostak light. In *Proceedings of the 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNAI*, pages 332–347, Copenhagen, Denmark, 2002. Springer.

[34] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In *IJCAR*, pages 67–82, 2008.

[36] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[37] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, pages 129–145, 2005.

[38] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48:1038–1068, September 2001.

[39] Wilfrid Hodges. *A shorter model theory*. Cambridge University Press, New York, NY, USA, 1997.

[40] SRI International. Yices 2.0. software, 2012.

[41] D. Kapur. A rewrite rule based framework for combining decision procedures. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FROCOS)*, volume 2309 of *LNAI*, pages 87–103, Santa Margherita Ligure, Italy, 2002. Springer.

[42] Daniel Kroening and Ofer Strichman. *Decision Procedures – an Algorithmic Point of View*. EATCS. Springer, 2008.

[43] Sava Krstić and Sylvain Conchon. Canonization for disjoint unions of theories. In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Computer Science*, Miami Beach, FL, USA, July 2003. Springer Verlag.

[44] Sava Krstić and Sylvain Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2):87–106, May 2005.

[45] Jean-Louis Lassez and Michael J Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9:373–379, 1992. 10.1007/BF00245296.

[46] Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq.* PhD thesis, January 2011.

[47] Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs 2008: In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[48] Stéphane Lescuyer and Sylvain Conchon. Improving coq propositional reasoning using a lazy cnf conversion scheme. In *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2009.

[49] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in Eddy. *STTT*, 11(1):13–25, 2009.

[50] R. Milner. A theory of type polymorphismn in programming. *Journal of Computer and System Sciences*, 17, 1978.

[51] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[52] Leonardo Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Leonardo Moura and Nikolaj Bjørner. Engineering dpll(t) + saturation. In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 475–490, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[55] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, 1980.

[56] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[57] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[58] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 82–97, London, UK, UK, 2001. Springer-Verlag.

[59] H. Rueß and N. Shankar. Deconstructing Shostak. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 19–28, Copenhagen, Denmark, 2001. IEEE Computer Society.

[60] N. Shankar and H. Rueß. Combining Shostak theories. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2378 of *LNCS*, pages 1–19, Copenhagen, Denmark, 2002. Springer.

[61] J.R. Shoenfield. *Mathematical logic*. Ak Peters Series. Association for Symbolic Logic, 1967.

[62] R. E. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, 1984.

[63] João P. Marques Silva and Karem A. Sakallah. Grasp, a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[64] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[65] Joao P. Marques Silva and Karem A. Sakallah. Grasp: a new search algorithm for satisfiability. In *ICCAD*, pages 220–227. IEEE Computer Society, 1996.

[66] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Proceedings of the First International Workshop on Frontiers of Combining Systems (FROCOS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, 1996.

[67] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.