Extracting Proofs from Branch-and-Prune

Sicun Gao **Soonho Kong** **Michael Wang**
Edmund M. Clarke

April 20, 2013
CMU-CS-13-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

δ-Complete decision procedures can solve SMT problems over the reals with a wide range of nonlinear functions, allowing "δ-bounded errors". The scalability of such procedures usually depends on efficient numerical procedures, whose implementation can be error-prone. It is important for δ-complete solvers to provide certificates to prove the correctness of their answers. We show how to do this for DPLL$\langle$ICP$\rangle$, a general solving framework based on Interval Constraint Propagation. We focus on the construction of proof trees for the "unsat" answers and the proof-checking of their correctness. Besides certifying solvers, we find our approach a promising one for automated theorem proving over the reals, exploiting the power of numerical algorithms in a formal way. One direct application is to establish many nonlinear lemmas in the Flyspeck project, for the formal proof of the Kepler Conjecture.

This research was sponsored by the National Science Foundation grants no. DMS1068829, no. CNS0926181 and no. CNS0931985, the GSRC under contract no. 1041377, the Semiconductor Research Corporation under contract no. 2005TJ1366, and the Office of Naval Research under award no. N000141010188.

Keywords: theorem proving, decision procedures, nonlinear theories of the reals

1 Introduction

SMT formulas over the real numbers can encode a wide range of problems in theorem proving and formal verification. Such formulas are very hard to solve when nonlinear functions are involved. Our recent work on δ-*complete decision procedures* provided a new general framework for handling nonlinear SMT problems over the reals [4, 5]. We say a decision procedure is δ-*complete* for a set S of SMT formulas, where δ is any positive rational number, if for any φ from S the procedure returns one of the following answers:

- unsat: φ is unsatisfiable.
- δ-sat: φ^δ is satisfiable.

Here, φ^δ is a syntactic variation of φ that encodes a notion of numerical perturbation on logic formulas (more details in Section 2). Essentially, we allow such a procedure to give answers with one-sided, δ-bounded errors. With this relaxation, δ-complete decision procedures can fully exploit the power of scalable numerical algorithms to solve nonlinear problems, and at the same time provide suitable correctness guarantees for many correctness-critical problems [4].

An important problem for practical SMT solvers is that the correctness of their answers should be verified. A standard approach is that, instead of complete verification of the software programs, a solver should provide certificates on-the-fly along with its answers. That is, when the solver determines that a formula $\varphi(\vec{x})$ is "sat", it produces an assignment $\vec{a}$ to all the variables such that the ground formula $\varphi(\vec{a})$ is easily checked to be true. On the other hand, when $\varphi(\vec{x})$ is determined to be "unsat", the solver can produce a proof P that establishes the validity of $\forall\vec{x}.\neg\varphi(\vec{x})$ in a suitable proof system. Here, P is called a *proof of unsatisfiability*. In the framework of δ-complete decision procedures, obtaining certificates from numerically-driven SMT solvers is especially important. Numerical algorithms usually contain complex heuristics and floating-point operations, and it is very hard to perform static verification on the programs directly. On the other hand, if the solutions witnessing "δ-sat" answers, and proofs of unsatisfiability for the "unsat" answers are extracted, we can check their correctness using stand-alone symbolic or arbitrary-precision computations. The inner mechanisms of the numerical algorithms are not relevant in the certification process.

From this perspective, our technique can be seen as a new approach to the challenging task of automated theorem proving over the reals. Note that the "unsat" answers never contains numerical errors. Such an approach would combine the best of two worlds: numerical procedures are fast but error-prone, and are used as oracles for the scalable exploration of the search space (either searching for a δ-solution, or a proof of unsatisfiability); symbolic algorithms are precise but slow, and are used for validating the outcome certificates, which is a much less computationally-intensive task.

In this paper, we show how to extract and validate such proofs of correctness for numerically-driven SMT solvers that implement the DPLL$\langle$ICP$\rangle$ algorithm [4], for solving nonlinear formulas over the reals. The challenge lies in extracting symbolic proofs of unsatisfiability that do not carry over the numerical errors, and the complete validation of them as theorems in a sound proof system (ideally, containing only simple proof rules).

Interval Constraint Propagation (ICP) [8] is a branch-and-prune algorithm for solving systems of real constraints, which acts as the theory solver in the DPLL(T) framework. The algorithm maintains an interval assignment to all the variables, and updates the assignments based on their consistency with the constraints. In a "pruning" step, ICP contracts the intervals by pruning away subintervals that do not contain any solution; in a "branching" step, ICP subdivides an interval to create subproblems and solve them recursively. The similarity between ICP and SAT solving techniques has been explored in the work [3].

Our approach is as follows. First, we formalize the ICP algorithm in the format of Abstract DPLL [13], such that its computation corresponds to sequences of abstract transitions. We then use a simple first-order proof calculus $\mathbb{D}_A$, relativized to a set A of axioms over the reals, and show how to transform a run of the Abstract ICP to a proof in the system. Next, we show how to validate the generated proofs using a stand-alone proof checker implementing simple rules and reliable interval arithmetic. The proof checker interacts with the solver in an abstraction refinement loop to obtain proof trees of sufficient detail. A main focus for our tool is to prove nonlinear lemmas in the Flyspeck project for the formal proof of the Kepler conjecture [7, 6]. They involve large numbers of nonlinear constraints including trigonometric functions. We show some promising experimental results towards the goal.

The paper is organized as follows. We review the framework of δ-complete decision procedures in Section 2. We then show how to formalize ICP, construct proofs, and proof-check them in Section 3. We report experiments with lemmas in the Flyspeck project in Section 4.

1.0.1 Related Work

Our work is closely related to several lines of research in the existing literature. For proving formulas with transcendental functions, MetiTarski [16, 1, 15] is the leading tool that reduces problems to polynomials and calls quantifier elimination procedures. For problems with only polynomials, Bernstein polynomials are used in PVS for formal proofs [12]. Our approach can be seen as complementary to these approaches. The iSAT solver [3] also contains strategies for certifying their answers in a different framework [10]. There are now several SMT solvers [9, 14] for formulas with nonlinear polynomials over the reals based on CAD with no proof-producing capacities, but a proof-producing algorithm is possible, as sketched in [11]. Proofs for correctness in general SMT solvers have been well studied recently in [18].

2 A Brief Review of δ-Complete Decision Procedures

We briefly review the framework of δ-complete decision procedures. It formulates a reasonable correctness requirement for decision procedures for nonlinear formulas over the reals. [19]).

First, to formalize computations over the reals, we need to encode the real numbers as infinite strings. We can then model computations of real functions with machines that can use infinite tapes as input and output. That is, a real function is computable if there exists a machine that computes, using oracles that encode the arguments of the function, the values of the function to an arbitrary precision. These notions are captured by the following definitions.

Definition 2.1 (Encoding Real Numbers [19]) *A* name *of a real number $a \in \mathbb{R}$ is a function $\gamma_a : \mathbb{N} \to \mathbb{Q}$ satisfying that for all $i \in \mathbb{N}$, $|\gamma_a(i) - a| < 2^{-i}$. For vectors $\vec{a} \in \mathbb{R}^n$, $\gamma_{\vec{a}}(i) = \langle \gamma_{a_1}(i), ..., \gamma_{a_n}(i) \rangle$. Write $\Gamma(\vec{a}) = \{\gamma : \gamma$ is a name of $\vec{a}\}$.*

Definition 2.2 (Computable Real Functions [19]) *We say $f :\subseteq \mathbb{R}^n \to \mathbb{R}$ is computable, if there exists an oracle Turing machine $\mathcal{M}_f$ as follows. Let $\vec{a} \in \mathrm{dom}(f)$ be any argument of f and $\gamma(\vec{a})$ any name of $\vec{a}$. On any input $i \in \mathbb{N}$, $\mathcal{M}_f^{\gamma(\vec{a})}(i)$ uses $\gamma(\vec{a})$ as an oracle, and computes a 2^{-i}-approximation to $f(\vec{a})$.*

Most common continuous real functions are computable, such as arithmetic, absolute value, $\min$, $\max$, $\exp$, $\sin$ and solutions of Lipschitz-continuous ordinary differential equations [19]. Compositions of computable functions are computable.

Now, suppose $\mathcal{F}$ denotes an arbitrary collection of computable real functions. Let $\mathcal{L}_{\mathcal{F}} = \langle <, \mathcal{F} \rangle$ be the first-order signature over the structure $\mathbb{R}_{\mathcal{F}} = \langle \mathbb{R}, \leq, \mathcal{F} \rangle$. (Constants are seen as 0-ary functions in $\mathcal{F}$.) We can then consider SMT problems over $\mathbb{R}_{\mathcal{F}}$, namely, satisfiability of quantifier-free $\mathcal{L}_{\mathcal{F}}$-formulas over $\mathbb{R}_{\mathcal{F}}$. We consider bounded SMT problems, which is more conveniently expressed as Σ_1-sentences with bounded quantifiers as follows. We say a Σ_1-sentence is bounded, if it can be written in the form

$$\varphi : \ \exists^{I_1} x_1 \cdots \exists^{I_n} x_n . \psi(x_1, ..., x_n)$$

where, for all i, $I_i \subseteq \mathbb{R}$ is a bounded (open or closed) interval; each bounded quantifier $\exists^{I_i} x_i . \phi$ denotes $\exists x_i . (x_i \in I_i \wedge \phi)$; $\psi(x_1, ..., x_n)$ is a quantifier-free $\mathcal{L}_{\mathcal{F}}$-formula, i.e., a Boolean combination of atomic formulas $f(x_1, ..., x_n) \circ 0$, where f is a composition of functions in $\mathcal{F}$ and $\circ \in \{<, \leq, >, \geq, =, \neq\}$. All the functions occurring in $\psi(\vec{x})$ should be defined everywhere over the closure of $I_1 \times \cdots \times I_n$. Any bounded Σ_1-sentence can be put into the following standard form, where inequalities are implicitly expressed by the bounds on quantifiers (using slack variables) and the atomic formulas only involve equalities:

Proposition 2.3 (Standard Form) *Any bounded Σ_1-sentence φ in $\mathcal{L}_{\mathcal{F}}$ is equivalent over $\mathbb{R}_{\mathcal{F}}$ to a sentence $\exists^{I_1} x_1 \cdots \exists^{I_n} x_n . \bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k_i} f_{ij}(\vec{x}) = 0)$.*

We can now define the notion of "δ-perturbations" on these formulas:

Definition 2.4 (δ-Weakening and Perturbations) *Let $\delta \in \mathbb{Q}^+$ be a constant and φ be a Σ_1-sentence in standard form:*

$$\varphi := \exists^{\vec{I}} \vec{x} \ (\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k_i} f_{ij}(\vec{x}) = 0)).$$

The δ-perturbed form (or δ-weakening) of φ defined as:

$$\varphi^{\delta} := \exists^{\vec{I}} \vec{x} \ (\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k} |f_{ij}(\vec{x})| \leq \delta)).$$

Also, a δ-perturbation is a constant vector $\vec{c} = (c_{11}, ..., c_{mk_m})$, where $c_{ij} \in \mathbb{R}$ and $||\vec{c}||_\infty \leq \delta$, such that $\varphi^{\vec{c}} := \exists^{\vec{I}} \vec{x} \ (\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k} f_{ij}(\vec{x}) = c_{ij}))$.

Definition 2.5 (Bounded δ-SMT in $\mathcal{L}_{\mathcal{F}}$) *Let $\mathcal{F}$ be a finite collection of Type 2 computable functions. Let φ be a bounded Σ_1-sentence in $\mathcal{L}_{\mathcal{F}}$ in standard form. The* bounded δ-SMT problem *asks for one of the following two decisions on φ:*

- unsat : *φ is false.*
- δ-sat : *φ^δ is true.*

When the two cases overlap, either decision can be returned.

The main theoretical result is that for any positive δ, the bounded δ-SMT problems in $\mathcal{L}_{\mathcal{F}}$ are decidable (namely, δ-decision procedures exist).

Theorem 2.6 (Decidability) *Let $\mathcal{F}$ be a finite collection of computable real functions and $\delta \in \mathbb{Q}^+$. The bounded δ-SMT problem in $\mathcal{L}_{\mathcal{F}}$ is decidable.*

The δ-SMT problems also have reasonable complexity bounds for various signatures that would otherwise define undecidable theories.

Theorem 2.7 (Complexity) *Let $\mathcal{F}$ be a finite collection of functions in Type 2 complexity class C, $\mathsf{P} \subseteq \mathsf{C} \subseteq \mathsf{PSPACE}$. The δ-SMT problem for uniformly bounded Σ_1-classes in $\mathcal{L}_{\mathcal{F}}$ is in NP^{C}. For instance, when $\mathcal{F}$ only contains P-time computable real functions such as $\{+, \times, \exp, \sin\}$, the problem is NP-complete.*

These results lead to a new perspective on decision problems over the reals in general. This framework provides a theoretical basis for the development of numerically-driven decision procedures.

3 Extracting Proofs from Interval Constraint Propagation

3.1 Formalizing Interval Constraint Propagation

Interval Constraint Propagation (ICP) [2] finds solutions of real constraints using the "branch-and-prune" method, combining interval arithmetic and constraint propagation. The idea is to use interval extensions of functions to "prune" out sets of points that are not in the solution set and "branch" on intervals when such pruning can not be done, recursively until a small enough box that may contain a solution is found or inconsistency is observed.

3.1.1 Abstract ICP

Our task now is to formalize ICP algorithms so that we can extract symbolic proofs from its computation process. The branch-and-prune structure of ICP is very similar to DPLL-based SAT solving, so we follow the format of Abstract DPLL. We will represent ICP as a transition system, whose states consist of interval assignments and the real constraints to be solved.

An interval I is any connected subset of $\mathbb{R}$, and we write $\mathbb{IR}$ to denote the set of all the intervals. We first formalize how ICP maintains interval assignments to a set of variables as follows.

Definition 3.1 (Interval Assignment Sequence) *Let $x_1, ..., x_n$ be real variables. An* interval assignment sequence *over $\vec{x}$ is a sequence $(s_1, ..., s_m)$, where*

$$s_i \in \{(x_i \in I_j) : 1 \leq i \leq n, I_j \in \mathbb{IR}\} \cup \{(x_i \in I_j)^d : 1 \leq i \leq n, I_j \in \mathbb{IR}\}.$$

We write (S_1, S_2) to denote the concatenation of two sequences S_1 and S_2. The parentheses can be omitted when appropriate.

It will be clear later that when we write $(x \in I)^d$, it means an arbitrary choice on the value of x (called a d-assignment), which is consequently a backtrack point.

Remark 3.2 *ICP can maintain unions of intervals for variables. In principle this is not needed if we only consider the decision problem, which only searches for one solution and the components of a union can be tested sequentially. So we assume that only connected subsets of values are used here.*

Definition 3.3 (Box Domain) *Let S be an interval assignment sequence over variables $x_1, ..., x_n$. The* box domain *associated with S is defined by*

$$\beta(S) = I_1 \times \cdots \times I_n, \text{ where } I_i = \bigcap \{I : (x_i \in I) \text{ or } (x_i \in I)^d \text{ occurs in } S\}.$$

Also, we write $\beta(S)_i$ to denote I_i.

Definition 3.4 (ICP Transitions) *Let $\vec{x} = (x_1, ..., x_n)$ be a vector real variables. We write $c(\vec{x})$ to denote an arbitrary constraint over $\mathbb{R}^n$, and S an interval assignment sequence over $\vec{x}$. Let $S \parallel c$ be the current state. We will always write $\beta(S_i) = I_i$ to denote the current interval assignment on variable x_i. We now define the following transition rules from $S \parallel c$ to another state.*

(Pruning) *Let I_i^1 be a subset of I_i such that $\forall \vec{a} \in \beta(S, x_i \in I_i^1)$, $c(\vec{a})$ is false. Then, if we let I_i^2 be an interval satisfying $I_i \subseteq I_i^1 \cup I_i^2$, then*

$$S \parallel c \quad \stackrel{p}{\Longrightarrow} \quad S, (x_i \in I_i^2) \parallel c$$

is called a pruning step.

(Branching) *Let I_i^1 be a subset of I_i. Then*

$$S \parallel c \quad \stackrel{br}{\Longrightarrow} \quad S, (x_i \in I_i^1)^d \parallel c,$$

is called a branching step.

(Backtracking) *Let I_i^1 be a subset of I_i, such that $\forall \vec{a} \in \beta(S, x_i \in I_i^1, S')$, $c(\vec{a})$ is false. Let I_i^2 be an interval such that $I \subseteq I_i^1 \cup I_i^2$. If in addition, S' does not contain any d-assignment (of the form $(x \in I)^d$), then we can make a transition*

$$S, (x_i \in I_i^1)^d, S' \parallel c \stackrel{bt}{\Longrightarrow} \quad S, (x_i \in I_i^2) \parallel c,$$

which is called a backtracking step.

(Failure) *Suppose $\forall \vec{a} \in \beta(S)$, $c(\vec{a})$ is false, and there is no d-assignment in S. Then we can make the transition*

$$S \parallel c \stackrel{f}{\Longrightarrow} \emptyset \parallel c$$

which is called a failure step.

Remark 3.5 *The main difference between the Abstract ICP and Abstract DPLL is that the assignments are not starting from empty, but contracted, and the interval assignments on variables can be nondeterministic.*

Definition 3.6 (Abstract ICP) *An n-dimensional ICP framework is a transition system*

$$\langle \mathbb{IR}^n, \mathcal{S}, \mathcal{C}, \Longrightarrow, \varepsilon \rangle$$

where $\mathcal{S}$ is the set of all interval assignment sequences over $\mathbb{IR}^n$, and $\mathcal{C}$ is any set of constraints over $\mathbb{R}^n$. A state is an element in $\mathcal{S} \parallel \mathcal{C}$. The transition rules $\Longrightarrow$: $\mathcal{S} \times \mathcal{C} \to \mathcal{S} \times \mathcal{C}$ are as specified in Definition 3.4. $\varepsilon \in \mathbb{Q}^+$ is an error bound. A run *of ICP is any sequence*

$$S_1 \parallel c, ..., S_k \parallel c,$$

where either S_k is $\emptyset$, or $S_k \neq \emptyset$ and $||\beta(S_k)|| < \varepsilon$.

Remark 3.7 *We have defined ICP in a general way, without enforcing conditions on the pruning operators, such as well-definedness. Thus, many invalid ICP runs can be generated. In this way, we treat ICP as a proof searching algorithm, and rely on the proof checkers to determine the correctness of an ICP run. In practice, of course, only "correct" ICP algorithms can provide proofs that can always be validated.*

Example 3.8 *Consider a constraint $c(x, y) : y = x \wedge y = x^2$, and $x \in [1.5, 2]$ and $y \in [1, 2]$ are the initial interval assignment. A possible ICP run is:*

$$\begin{aligned}
x \in [1.5,2], y \in [1,2] \parallel c \quad &\stackrel{br}{\Longrightarrow} \quad x \in [1,2], y \in [1,2], (x \in [1.7,2])^d \parallel c \\
&\stackrel{bt}{\Longrightarrow} \quad x \in [1,2], y \in [1,2], x \in [1.5,1.7] \\
&\qquad \textit{(backtracking, since } \forall \vec{a} \in [1.7,2] \times [1,2], c(\vec{a}) \textit{ is false,} \\
&\qquad \textit{and } [1.5,2] \subseteq [1.5,1.7] \cup [1.5,2] \textit{ for } x) \\
&\stackrel{p}{\Longrightarrow} \quad x \in [1,2], y \in [1,2], x \in [1.5,1.7], x \in [1.5,1.6] \parallel c \\
&\qquad \textit{(pruning, since } \forall \vec{a} \in [1.6,1.7] \times [1,2], c(\vec{a}) \textit{ is false)} \\
&\stackrel{p}{\Longrightarrow} \quad x \in [1,2], y \in [1,2], x \in [1.5,1.7], x \in [1.5,1.6], x \in \emptyset \parallel c \\
&\qquad \textit{(pruning, since } \forall \vec{a} \in [1.5,1.6] \times [1,2], c(\vec{a}) \textit{ is false)} \\
&\stackrel{f}{\Longrightarrow} \quad \emptyset || c \ \textit{(since } \forall \vec{a} \in \emptyset \times [1,2], c(\vec{a}) \textit{ is false.)}
\end{aligned}$$

3.2 First-Order Proofs of Unsatisfiability

We focus on the proof the unsatisfiability of conjunctions of theory atoms in the DPLL(T) framework, i.e., formulas of the form

$$\exists^{I_1} x_1 \cdots \exists^{I_n} x_n. \bigwedge_{i=1}^{m} f_i(x_1, ..., x_n) \sim 0$$

where $\sim \in \{=, \neq, >, \geq, <, \leq\}$. It is clear that once such proofs are obtained, the proof of unsatisfiability of Boolean combinations of the theory atoms can be obtained, by simply plugging them in the high level resolution proof. Also, it is important to note that the ICP algorithm solves *systems* of constraints, and it regards the conjunction $\bigwedge_{i=1}^{m} f_i(x_1, ..., x_n) \sim 0$ as one constraint $c(x_1, ..., x_n)$. Consequently, our task is now reduced to obtaining proofs for the validity of formulas of the form $\forall x_1 \cdots \forall x_n.(x_1 \in I_1 \wedge \cdots \wedge x_n \in I_n) \rightarrow \neg c(\vec{x})$, from the failure of ICP search for a solution to the original SMT formula $\exists \vec{x}.\vec{x} \in \vec{I} \wedge c(\vec{x})$.

We will construct a simple first-order proof calculus, and show how to transform ICP runs into proofs in the system.

Again, we consider formulas in a signature $\mathcal{L}_F = \langle <, \mathcal{F} \rangle$, where constants are considered as 0-ary functions in $\mathcal{F}$. When we write $x \in I$, where I denotes an interval, it is regarded as an abbreviation for their equivalent $\mathcal{L}_\mathcal{F}$-formula. Note that this means that I only uses $\mathcal{L}_\mathcal{F}$-terms as end-points. Also, as mentioned above, $c(\vec{x})$ abbreviates a conjunction of atomic formulas. We also allow the use of vectors in the formulas, writing $\vec{x} \in \vec{I}$ to denote $\bigwedge_i x_i \in I_i$.

Definition 3.9 (System $\mathbb{D}_A$) *We define $\mathbb{D}_A$ to be the first-order proof system consisting of only the following two rules:*

$$\frac{\forall \vec{x}(\psi \rightarrow \varphi) \qquad \forall \vec{x}(\psi' \rightarrow \varphi)}{\forall \vec{x}(\psi \vee \psi' \rightarrow \varphi)} \vee I \qquad\qquad \frac{\forall x(\psi \rightarrow \varphi) \qquad \forall x(\psi' \rightarrow \psi)}{\forall x(\psi' \rightarrow \varphi)} \forall MP$$

and a set A of axioms of the following two types:

Interval Axioms

$$\frac{}{\forall \vec{x}(\vec{x} \in \vec{I} \rightarrow \vec{x} \in \vec{I_1} \vee \vec{x} \in \vec{I_2})} IA$$

Constraint Axioms

$$\frac{}{\forall \vec{x}(\vec{x} \in \vec{I} \rightarrow c(\vec{x}))} CA$$

Derivations in $\mathbb{D}_A$ are as standardly defined, as natural deductions following these rules. Clearly, the two first-order rules are valid. Thus, if all the axioms in A are valid, then the system only produces valid formulas over $\mathbb{R}$.

Proposition 3.10 (Soundness) *If $\mathbb{D}_A \vdash \varphi$ and $\mathbb{R} \models \bigwedge A$, then $\mathbb{R} \models \varphi$.*

Remark 3.11 *Clearly, the constraint axioms are the most nontrivial part. They are the basic facts of real functions that a numerical procedure relies on, usually concerning the range of functions within a small interval. The interval axioms are sometimes not trivial either (for instance, compare intervals ending with e^π and π^e respectively). Proof-checking involves validation of these axioms, which we discuss in Section 3.3.*

We now describe the construction of proof trees from ICP runs, which will be represented as labeled binary trees. A labeled binary tree is defined as a tuple $T = \langle V, V_L, \Sigma, \delta, \sigma \rangle$. Here, $V = \{v_0, ..., v_k\}$, is a finite set of nodes, where $v_0 \in V$ always denotes the root node. Σ is a set of labels, which in our case is the set of $\mathcal{L}_\mathcal{F}$-formulas. $\delta :\subseteq V \times \{l, r\} \to V$ is a partial mapping from a node to its descendant nodes, where $\delta(v, l)$ and $\delta(v, r)$ denote the left and right descendant nodes, respectively. $\sigma :\subseteq V \to \Sigma$ is a labeling function that maps each node $v \in V$ to a formula $\sigma(v) \in \Sigma$. In addition, the edges in the tree can be labeled as well, through a function $\tau : V \times V \to \Omega$ where Ω is a set of edge-labels.

3.2.1 Tree Generation

Let an ICP run be

$$S_0 \parallel c \overset{t_1}{\Longrightarrow} \cdots \overset{t_m}{\Longrightarrow} S_m \parallel c,$$

such that the ending transition t_m is a failure step, i.e., $S_m = \emptyset$. We now define the procedure by defining the functions δ and V_L through induction on s_i. The edges can be labeled naturally with Ω = {$\vee$I, $\forall$M, IA, CA}.

Case $i = 0$. We label the root node v_0 by

$$\sigma(v_0) := \forall \vec{x}(\vec{x} \in \beta(S_0) \to \neg c).$$

Let $V_L^0 = \{\delta(v_0, l), \delta(v_0, r)\}$ denote the current collection of leaf nodes. Note that this formula is the negation of the input SMT formula.

Case $i = k + 1$ ($1 < k \leq m$). Suppose V_L^k and σ have been defined for $s_1, ..., s_k$. Write $s_k = S_k \parallel c$ and $s_{k+1} = S_{k+1} \parallel c$. Now we split the cases on the type of the step t from s_k to s_{k+1} as follows. Again, we use the convention that $\beta(S)_i = I_i$ denotes the current interval assignment on a variable x_i.

(Pruning Case) Suppose $s_k \Longrightarrow s_{k+1}$ is a pruning step. That is,

$$S_k \parallel c \overset{p}{\Longrightarrow} S_k, (x_i \in I_i^2) \parallel c,$$

where $I_i \subseteq I_i^1 \cup I_i^2$ and $\forall \vec{a} \in \beta(S_k, x_i \in I_i^1)$, $c(\vec{a})$ is false. If we write

$$\vec{I_1} = \beta(S_k, (x_i \in I_i^1)), \vec{I_2} = \beta(S_k, (x_i \in I_i^2)), \text{ and } \vec{I} = \beta(S_k),$$

then this step corresponds to the following sub-tree of the proof

$$\dfrac{\dfrac{\dfrac{\vdots}{\forall\vec{x}(\vec{x}\in\vec{I}_2\to\neg c)}\qquad\dfrac{}{\forall\vec{x}(\vec{x}\in\vec{I}_1\to\neg c)}\text{CA}}{\forall x((\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2)\to\neg c)}\vee\text{I}\qquad\dfrac{}{\forall x(x\in I_i\to(x\in I_i^1\vee x\in I_i^2))}\text{IA}}{\forall\vec{x}(\vec{x}\in\vec{I}\to\neg c)}\forall\text{M}$$

Formally, the sub-tree is added as follows. Let $v\in V_L^k$ be an existing leaf node that is labeled by the formula corresponding to $S_k \parallel c$; namely,

$$\sigma(v)=\forall\vec{x}(\vec{x}\in\vec{I}\to\neg c).$$

(We will inductively prove that such a node exists.) We then define

$$\begin{aligned}
\delta(v,l) &= v_{k+1}^1,\sigma(v_{k+1}^1)=\forall\vec{x}((\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2)\to\neg c);\\
\delta(v,r) &= v_{k+1}^2,\sigma(v_{k+1}^2)=\forall\vec{x}(\vec{x}\in\vec{I}\to(\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2));\\
\delta(v_{k+1}^1,l) &= v_{k+1}^3,\sigma(v_{k+1}^3)=\forall\vec{x}(\vec{x}\in\vec{I}_2\to\neg c)\\
\delta(v_{k+1}^1,r) &= v_{k+1}^4,\sigma(v_{k+1}^4)=\forall\vec{x}(\vec{x}\in\vec{I}_1\to\neg c)
\end{aligned}$$

and set $V_L^{k+1}=(V_L^k\setminus\{v\})\cup\{v_{k+1}^2\}$.

(Branching Case) Suppose $s_k\Longrightarrow s_{k+1}$ is a branching step. That is,

$$S_k\parallel c\ \overset{br}{\Longrightarrow}\ S_k,(x_i\in I_i^1)^d\parallel c,$$

under the condition that $I_i^1\subseteq I_i$. If we write

$$\vec{I}_1=\beta(S,(x_i\in I_i^1)),\vec{I}_2=\beta(S,(x_i\in I_i^2)),\text{ and }\vec{I}=\beta(S),$$

where $I\subseteq I_i^1\cup I_2$, then this step corresponds to the following sub-tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{\forall\vec{x}(\vec{x}\in\vec{I}_1\to\neg c)}\qquad\dfrac{\vdots}{\forall\vec{x}(\vec{x}\in\vec{I}_2\to\neg c)}}{\forall x(\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2\to\neg c)}\vee\text{I}\qquad\dfrac{}{\forall x(x\in I_i\to(x\in I_i^1\vee x\in I_i^2))}\text{IA}}{\forall\vec{x}(\vec{x}\in\vec{I}\to\neg c)}\forall\text{M}$$

Formally it is defined as follows. Again, let $v\in V_L^k$ be a leaf node such that $\sigma(v)=\forall\vec{x}(\vec{x}\in\vec{I}\to\neg c)$. We then define

$$\begin{aligned}
\delta(v,l) &= v_{k+1}^1,\sigma(v_{k+1}^1)=\forall\vec{x}(\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2\to\neg c);\\
\delta(v,r) &= v_{k+1}^2,\sigma(v_{k+1}^2)=\forall\vec{x}(\vec{x}\in\vec{I}\to(\vec{x}\in\vec{I}_1\vee\vec{x}\in\vec{I}_2));\\
\delta(v_{k+1}^1,l) &= v_{k+1}^3,\sigma(v_{k+1}^3)=\forall\vec{x}(\vec{x}\in\vec{I}_1\to\neg c)\\
\delta(v_{k+1}^1,r) &= v_{k+1}^4,\sigma(v_{k+1}^4)=\forall\vec{x}(\vec{x}\in\vec{I}_2\to\neg c)
\end{aligned}$$

and set $V_L^{k+1}=(V_L^k\setminus\{v\})\cup\{v_{k+1}^3,v_{k+1}^4\}$.

(Backtracking Case) Suppose $s_k \Longrightarrow s_{k+1}$ is a branching step. That is,

$$S_{k'}, (x_i \in I_i)^d, S' \parallel c \quad \overset{bt}{\Longrightarrow} \quad S_{k'}, (x_i \in I_i^2) \parallel c,$$

when $\forall a \in \beta(S, (x_i \in I_i)^d, S')$, $c(\vec{a})$ is false, and $I_i \subseteq I_i^2 \cup I_i^1$, where $I_i = \beta(S_{k'})_i$. $S_{k'}$ is a previous interval assignment sequence, with $k' < k$. If we write

$$\vec{I_1} = \beta(S, (x_i \in I_i^1)), \vec{I_2} = \beta(S, (x_i \in I_i^2), \text{ and } \vec{I} = \beta(S_{k'}),$$

then this step corresponds to the following sub-tree of the proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall \vec{x}(\vec{x} \in \beta(S_{k'}, (x \in I_i^1)^d, S') \to \neg c)}\text{CA}}{\vdots}}{\forall \vec{x}(\vec{x} \in \vec{I_1} \to \neg c)} \qquad \dfrac{\vdots}{\forall \vec{x}(\vec{x} \in \vec{I_2} \to \neg c)}}{\forall x(\vec{x} \in \vec{I_1} \vee \vec{x} \in \vec{I_2} \to \neg c)}\vee\text{-I} \qquad \vdots}{\forall \vec{x}(\vec{x} \in \vec{I} \to \neg c)}\forall\text{-MP}$$

Formally, we simply set $V_L^{k+1} = V_L^k$, and do not update the labeling.

(Fail Case) Suppose it is a failure step. That is,

$$S \parallel c \quad \overset{f}{\Longrightarrow} \quad \emptyset \parallel c$$

when $\forall \vec{a} \in \beta(S)$, $c(\vec{a})$ is false and S has no d-assignments. Let $\vec{I} = \beta(S)$. This step corresponds to

$$\dfrac{}{\forall \vec{x}(\vec{x} \in \vec{I}) \to \neg c}\text{FA}$$

We set $V_L^{k+1} = V_L^k$ and do not update σ.

Complete tree. In all, after all the steps in the ICP run are followed, the tree that we construct is $T = \langle V, V_L^m, \delta, \sigma \rangle$. The axiom set is given by

$$A = \{\sigma(v) : v \in V_L^m\}.$$

It is easy to see that T is a valid proof tree in $\mathbb{D}_A$.

Proposition 3.12 *For every ICP run ending with $\emptyset \parallel c$, the tree construction procedure above produces a valid natural deduction tree in $\mathbb{D}_A$.*

Proposition 3.13 *The size of the proofs is linear in the computation steps, which can be exponential in the size of the problems.*

Again, once the proof tree is constructed, the details of the ICP algorithm no longer matters. The only rules involved are the two first-order rules in $\mathbb{D}_A$. Following relative soundness of the system, to establish validity of the formula, we only need to validate the axiom set A.

3.3 Validating the Axioms

There are two types of axioms that we allow in the proofs constructed from ICP runs: interval axioms and constraint axioms. To validate such axioms, we still need numerical computations. However, we will show that they only rely on simple computations that can be validated through stand-alone arbitrary-precision or symbolic computation. Note that the validation of the axioms can fail – they can fail even when the solver correctly returns unsat, if the solver uses advanced numerical heuristics leading to axioms that can not be verified by reliable numerical computation. In practice, we ensure the correctness of the proof checker first, and use an abstraction refinement loop that allows the proof checker to ask for more detailed proofs from the solver.

The interval axioms do not contain any real functions, and are of the form $\forall \vec{x}(x \in I_1 \vee x \in I_2 \rightarrow x \in I)$. We only need to check that I is a subset of $I_1 \cup I_2$ by comparing the end points of the intervals, which is an easy numerical task.

The constraint axioms are of the form $\forall x(\vec{x} \in \vec{I} \rightarrow c(\vec{x}))$, and can only be verified by considering the functions that occur in c. Although they are of the same form as the formulas we solve, these axioms should contain evident properties of the functions involved, usually on small intervals. Such facts can be verified using reliable interval computations, for instance as follows.

Definition 3.14 (Interval Extensions) *Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a real function. An interval function $F : \mathbb{IR}^n \rightarrow \mathbb{IR}$ is a function that satisfies:*

$$\forall I \in \mathrm{dom}(F), \{f(x) : x \in I\} \subseteq F(I).$$

A simple example of interval extensions is the *natural interval extension* for arithmetic operations, based on computations of functions on the end points of intervals.

Proposition 3.15 *Let F be an interval extension of f, and $I \subseteq \mathrm{dom}(f)$. If $F(I) \subseteq A$, then $\forall x(x \in I \rightarrow f(x) \in A)$.*

Thus, the axioms are validated if we can verify that they are consistent with all the interval extensions.

Example 3.16 *The second pruning step in Example 3.8 generates an axiom*

$$\forall x \forall y(x \in [1.7, 2] \wedge y \in [1, 2] \rightarrow \neg(y = x^2) \vee \neg(y = x))$$

This can be easily validated through the natural interval extension of $(y - x^2)$, which is $[1, 2] - [1.7, 2]^2 = [-3, -0.89]$ and does not contain 0.

In practice, ICP usually implements complicated heuristics that are more powerful than what can be verified through direct interval arithmetic. In principle, we could simulate the numerical heuristics to a certain extent with reliable computations. However, a more practical approach first is to use an abstraction refinement loop that allows the proof checker to ask the solver for proof traces of the right amount of details. We call this the "Branch and Prove" loop.

When we fail to prove an axiom through simple interval arithmetic, the proof checker generates new subproblems that are returned to the solver. At this stage, the axioms becomes the new

theorems to be proved. This is an abstraction refinement procedure. By executing the loop, we obtain proof trees that contain more and more detailed steps. There are two ways that the prover can generate the subproblems, branching on a variable in the formula or using a smaller δ. Note that under the condition that the pruning operators in the solver is well-defined, both procedures never change the unsat result. The branching may give exponentially many new problems; while the δ-change does not give new problems, but may exponentially slow down the solver in each round. In practice we observe that such a refinement loop is very useful, as we will show in the experiments.

4 Case Study: Proving Lemmas for the Kepler Conjecture

We build the proof-producing capacity into our open-source tool dReal[1]. All the experiments below are run on a machine of with a 32-core 2.3GHz AMD Opteron Processor and 94GB of RAM. A

Problem#	#OP	$\mathrm{Time_S}$	Result	Trace Size	PC	#PA	#SP	$\mathrm{Time_{PC}}$	#D
489	60	0:00.51	UNSAT	545	O	2	1	0:01.44	1
50	33	0:00.42	UNSAT	14,029	O	7	4	0:02.26	2
51	33	0:00.33	UNSAT	11,701	O	5	3	0:02.21	2
52	33	0:00.08	UNSAT	62,539	O	47	30	0:04.33	3
48	33	0:01.45	UNSAT	506,453	O	432	325	0:24.86	4
53	17	0:00.38	UNSAT	81,132	O	146	133	0:15.64	9
54	17	0:00.26	UNSAT	3,052	O	41	40	0:11.35	9
55	17	0:00.39	UNSAT	75,675	O	76	64	0:09.74	9
172	17	0:00.15	UNSAT	221	O	77	76	0:10.08	9
164	16	0:02.23	UNSAT	1,102,561	O	715	508	0:23.56	4
101	595	3:27.63	UNSAT	24,070,451	X	——	——	———	–
142	576	9:05.64	UNSAT	14,385,073	X	——	——	———	–
45	56,120	22:13.15	UNSAT	5,678,790	X	——	——	———	–
44	1,353,167	N/A	TO	N/A	–	——	——	———	–
46	31,594	N/A	TO	N/A	–	——	——	———	–
43	907,591	N/A	TO	N/A	–	——	——	———	–
42	908,008	N/A	TO	N/A	–	——	——	———	–

Table 1: Experimental results. #OP = Number of nonlinear operators in the problem, $\mathrm{TIME_S}$ = Solving time in seconds, TO = Timeout (30min), PC = Proof Checked, #PA = Number of proved axioms, #SP = Number of subproblems generated by proof checking, $\mathrm{TIME_{PC}}$ = Proof-checking time in seconds, #D = Number of iteration depth required in proof checking.

main motivation for us to build the proof checker is to contribute to the Flyspeck project [6],

[1]Links to the tool and benchmarks are on our homepages, `http://www.cs.cmu.edu/~sicung` and `http://www.cs.cmu.edu/~soonhok`

for the fully formalized proof of the Kepler conjecture. As lemmas for the proof, hundreds of nonlinear real inequalities need to be verified. Although the formulas usually contain only around ten variables, they contain a huge number of nonlinear arithmetic operations and trigonometric functions, and are mathematically challenging. The following is typical:

$$\forall \vec{x} \in [2, 2.51]^6.\ \Big(-\frac{\pi - 4\arctan\frac{\sqrt{2}}{5}}{12\sqrt{2}}\sqrt{\Delta(\vec{x})} + \frac{2}{3}\sum_{i=0}^{3}\arctan\frac{\sqrt{\Delta(\vec{x})}}{a_i(\vec{x})} \leq -\frac{\pi}{3} + 4\arctan\frac{\sqrt{2}}{5}\Big)$$

where $a_i(\vec{x})$ are quadratic and $\Delta(\vec{x})$ is the determinant of a nonlinear matrix.

In the original proof, Hales implemented procedures that combine linear programming and interval arithmetic to establish all these formulas, but the algorithms are hard to be formally verified. In fact, the formal verification of these formulas is the last main piece of work needed to complete the full project. The state of the art is explained in a very recent thesis [17], reporting the proofs of about 10 inequalities so far, using formal Taylor series and interval arithmetic.

Without any particular optimization on ICP, we have already observed promising results. Out of a total number of 505 nonlinear formulas in the Flyspeck project repository, we solved 137 of them returning unsat with a timeout of 30 minutes and $\delta = 10^{-3}$. Out of these formulas, we applied the proof checking algorithm, and formally proved 55 of them directly.The proof traces of these formulas can be very large; for instance, we proved one with 47k lines in the proof (1MB file). In Table 1, we list some of the representative benchmarks to show scalability. A full table for all the results is on the tool page. It also contains results for other standard benchmarks, such as from [9].

As the proof-generation capacity is for debugging SMT solvers, the proofs have also been valuable for us to observe bugs during the process. For instance, we have observed unstable behavior in the trigonometric function evaluation in realpaver (the cosine function around $4\pi/3$). On the other hand, our proof checker only involves simple operations written in OCaml, and performs the checking completely independently from the solver. Thus, for the unsat answers with proofs, we are confident about the correctness of the answer, which does not involve bugs from realpaver. Ultimately, we aim for having a verified proof checker in systems such as HOL.

5 Conclusion

We presented our approach for extracting formal proofs from a numerically-driven decision procedure in the DPLL$\langle$ICP$\rangle$ framework. We formalized the ICP algorithm, and showed how to validate proof trees from the unsat answers. A main focus for our tool is to prove nonlinear lemmas in the Flyspeck project, and we have observed promising experimental results. We believe the approach can be combined with exiting symbolic methods. We regard our work as a first step in a promising approach towards the formal verification of the nonlinear lemmas in the Flyspeck project. Further work would involve proof abstractions, local heuristics, and an implementation of our proof checker in HOL.

References

[1] B. Akbarpour and L. C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3):175–205, 2010.

[2] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16. Elsevier, 2006.

[3] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.

[4] S. Gao, J. Avigad, and E. M. Clarke. Delta-complete decision procedures for satisfiability over the reals. In *IJCAR*, pages 286–300, 2012.

[5] S. Gao, J. Avigad, and E. M. Clarke. Delta-decidability over the reals. In *LICS*, pages 305–314, 2012.

[6] T. C. Hales. Introduction to the flyspeck project. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[7] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[8] P. V. Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, 1997.

[9] D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In *IJCAR*, pages 339–354, 2012.

[10] S. Kupferschmid, B. Becker, T. Teige, and M. Fränzle. Proof certificates and non-linear arithmetic constraints. In R. Kraemer, A. Pawlak, A. Steininger, M. Schölzel, J. Raik, and H. T. Vierhaus, editors, *DDECS*, pages 429–434. IEEE, 2011.

[11] S. McLaughlin and J. Harrison. A proof-producing decision procedure for real arithmetic. In *CADE*, pages 295–314, 2005.

[12] C. Muñoz and A. Narkawicz. Formalization of a representation of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 2012. Accepted for publication.

[13] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[14] G. O. Passmore and P. B. Jackson. Combined decision techniques for the existential theory of the reals. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2009.

[15] G. O. Passmore, L. C. Paulson, and L. M. de Moura. Real algebraic strategies for metitarski proofs. In J. Jeuring, J. A. Campbell, J. Carette, G. D. Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *AISC/MKM/Calculemus*, volume 7362 of *Lecture Notes in Computer Science*, pages 358–370. Springer, 2012.

[16] L. C. Paulson. Metitarski: Past and future. In L. Beringer and A. P. Felty, editors, *ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2012.

[17] A. Solovyev. Formal computations and methods. PhD Thesis, University of Pittsburgh, 2012.

[18] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[19] K. Weihrauch. *Computable Analysis: An Introduction*. 2000.