Proofs and Refutations, and Z3

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research

Abstract

Z3 [3] is a state-of-the-art Satisfiability Modulo Theories (SMT) solver freely available from Microsoft Research. It solves the decision problem for quantifier-free formulas with respect to combinations of theories, such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is used in various software analysis and test-case generation projects at Microsoft Research and elsewhere. The requirements from the user-base range from establishing validity, dually unsatisfiability, of first-order formulas; to identify invalid, dually satisfiable, formulas. In both cases, there is often a need for more than just a yes/no answer from the prover. A model can exhibit why an invalid formula is not provable, and a proof-object can certify the validity of a formula. This paper describes the proof-producing internals of Z3. We also briefly introduce the model-producing facilities. We emphasize two features that can be of general interest: (1) we introduce a notion of implicit quotation to avoid introducing auxiliary variables, it simplifies the creation of proof objects considerably; (2) we produce natural deduction style proofs to facilitate modular proof re-construction.

1 Introduction

The title of our paper borrows from Imre Lakatos's famous book on conjectures, proofs and refutations in informal mathematics [7], yet our setting is machine checked proofs, that are penultimately given in a context of formal systems where proofs are derived from axioms. Proofs in our context are derivations from axioms, or derivations in theories that have solvers, implemented using efficient algorithms that need only produce derivations implicitly. Part of the challenge is that efficiently and compact checkable proofs, or certificates, are to be extracted from the solvers. Refutations, also called models, are counter-examples, that exhibit interpretations for formulas that do not follow from asserted axioms. Models are also extracted from solvers.

Applications of SMT solvers that consume models benefit tremendously from a prover that produces more than just a yes/no answer or a set of saturated clauses. A solver should therefore be able to communicate a model that can be represented finitely and consumed by the clients. Applications that mainly require an indication of validity may in some cases furthermore benefit from a certificate in the form of a proof object. This paper describes the model-producing features of Z3, currently available in the tool. It also describes the proof-producing features in preparation in the next version of Z3. While this paper can serve as an overview of the model- and proof-producing facilities in Z3, we point out the following particularities of our approach:

1. We define a notion of implicit quotation that allows us to encode a Tseitsin' style clausification without introducing auxiliary symbols. Other proof-producing SMT systems that we are aware of [18], [19], [2], [22], introduce auxiliary symbols (such as proxy literals) during clausification and other transformations. Such symbols can impede optimizations in the theory solvers (we provide an example in Section 3.3.2 where we can avoid introducing extra Simplex Tableaux rows) and make proof re-construction more involved.

2. We adapt an open-ended architecture for representing proofs. Proof rules can be introduced by respective theory modules and combined with others. At the propositional level this is manifested as we adapt a natural deduction style calculus. This contrasts with existing proof-producing SAT solvers that generate resolution proofs directly [10, 6, 21]. We are obviously not the first to use natural deduction in the context of SMT, for example, [9], investigates efficient proof checking of natural deduction style proofs by implementing inference rules as rewrites.

3. We also do not attempt to specify all inference rules from a smaller set of axioms. Instead we rely on proof checking to be able to carry out a limited set of inferences, or refine proofs in a separate pass. This choice obviously reflects a trade-off between the requirements on the solver vs. the proof checker. Our own experience has been that the coarse granularity has in fact been sufficient in order to catch implementation bugs. Future work includes investigating whether this approach is practical in the context of proof checkers based on trusted cores [1].

2 Preliminaries

2.1 Terms and Formulas

Z3 uses basic multi-sorted first-order terms. Formulas are just terms of Boolean sort, and terms are built by function application, quantification, and bound variables. Sorts range over a finite denumerable set of disjoint primitive sorts. To summarize

$$\begin{array}{lllll} s & \in & \mathit{Sorts} & ::= & \mathsf{Boolean} \mid \mathsf{Int} \mid \mathsf{Proof} \mid \ldots & \\ t & \in & \mathit{Terms} & ::= & f(t_1, \ldots, t_n) & \text{function application} \\ & & & \mid & x & \text{bound variable} \\ & & & \mid & \forall x : s \,.\, t \;\mid\; \exists x : s \,.\, t & \text{quantification} \end{array}$$

There a a few built-in sorts, such as $\mathsf{Boolean}$, Int, Real, $\mathsf{BitVec}[n]$ (for each n, an n-bit bit-vector), and Proof. The Proof sort is used for proof-terms. Terms can be annotated by pragmas. For example, quantifiers are annotated with patterns that control quantifier instantiation. Function symbols can be both interpreted and uninterpreted. For example, numerals are encoded using interpreted functions. Function symbols also have attributes, such as to indicate whether they are associative and/or commutative. Note that there are no binding operators other than universal and existential quantification. A number of function symbols are built-in to the base theory. We will introduce the set of proof-constructing terms as we explain their origination, but here let us summarize the main pre-declared function symbols. We use the usual infix symbols $\wedge, \vee, \rightarrow, \leftrightarrow$ for Boolean conjunction, disjunction, implication and bi-implication. For each sort s there is an equality relation $\simeq$: $s \times s \rightarrow \mathsf{Boolean}$. The relation $\sim$: $\mathsf{Boolean} \times \mathsf{Boolean} \rightarrow \mathsf{Boolean}$ is used in proof terms. A proof of $\varphi \sim \psi$ establishes that φ is equisatisfiable with ψ. In other words, if we close φ and ψ by second-order existential quantification of all Skolem functions and constants, we obtain equivalent formulas.

2.2 Proof Terms

Proof objects are also represented as terms. So a proof-tree is just a term where each inference rule is represented by a function symbol. For example, consider the proof-rule for modus ponens:

$$\mathsf{modus\_ponens}\ \frac{\begin{array}{c}\vdots\, p \\ \psi \rightarrow \varphi\end{array} \qquad \begin{array}{c}\vdots\, q \\ \psi\end{array}}{\varphi}$$

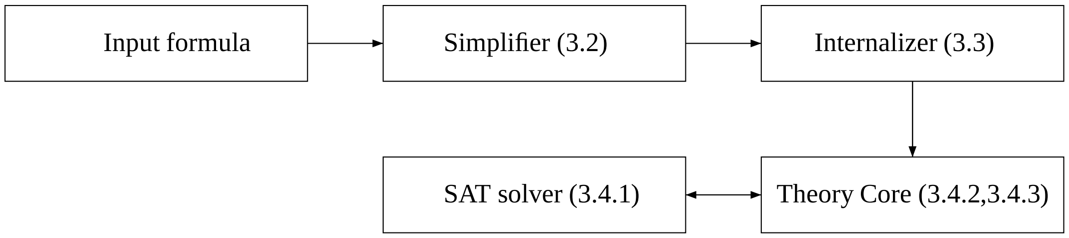
In the rule, p is a proof-term for $\psi \rightarrow \varphi$, and q is a proof for ψ. The resulting proof-term for φ is then $mp(p, q, \varphi)$. We will later elaborate on the function symbols for building basic proof terms that are available in Z3.

Every proof term has a *consequent*, the formula that the proof establishes. It is always the last argument in our proof terms. To access the consequent we will use the notation $con(p)$. For example, $con(mp(p, q, \varphi)) = \varphi$.

3 Proofs

3.1 Overview

Z3 applies multiple stages when attempting to verify a formula. First formulas are simplified using a repository of simplification rules. The result of simplification is then converted into an internal format that can be processed by the solver core. We refer to this phase as *internalization*. The core comprises of a SAT solver that performs Boolean search and a collection of theory solvers. The solver for equality and uninterpreted function symbols (congruence closure) features predominantly as a theory solver in Z3 as it dispatches constraints between the SAT solver and other theories.



As the figure illustrates, we will describe the various modules in the following sections, as we introduce the proof terms that are produced as a side effect of the modules.

The figure does not reflect the full extent of Z3. We will not be elaborating on proof objects for non-ground formulas in this paper. Z3 does produce proof objects for non-ground formulas. The required machinery introduces judgments stating equi-satisfiability of formulas. Furthermore, Z3 contains a module that produces and integrates proofs in a superposition calculus [4]. The proof terms are the usual superposition inferences [12]. That material is beyond the scope of this paper.

3.2 Simplification Rewriting

In a first phase, formulas are simplified using a rewriting simplifier. The simplifier applies standard simplification rules for the supported theories. For example, terms using the arithmetical operations, both for integer, real, and bit-vector arithmetic, are normalized into sums of monomials. A single axiom called $\mathit{rewrite}$ is used to record the simplification steps.

$\mathit{rewrite}(t \simeq s)$, $\mathit{rewrite}(\varphi \leftrightarrow \psi)$ A proof for a local rewriting step that converts t to s or φ to ψ. The head function symbol of t is interpreted. Sample instances of this proof object are: $\mathit{rewrite}(x+0 \simeq x)$, $\mathit{rewrite}(x + x \simeq 2 \cdot x)$, and $\mathit{rewrite}((\varphi \vee \texttt{false}) \leftrightarrow \varphi)$.

Notice that we do not axiomatize the legal rewrites. Instead, to check the rewrite steps, we rely on a proof checker to be able to apply similar inferences for the set of built-in theories: arithmetic, bit-vectors and arrays.

3.3 Internalization

Internalization is the process of translating an arbitrary formula φ into a normal form that can be consumed by efficient proof-search procedures. We will discuss two internalizations: clausification and conversion of arithmetical constraints into a format that can be processed using a global Simplex tableau.

Internalization often introduces auxiliary variables that are satisfiability preserving definitional extensions of the original problem. We will introduce a simple, but apparently unrecognized, technique of *implicit quotation* to allow us introducing these definitional extensions, but at the same time take advantage of the fact that the auxiliary variables can be viewed directly as terms they are shorthand for. The technique of implicit quotation makes the proof extraction process fairly direct.

3.3.1 Clausification

Tseitsin's clausal form conversion algorithm can be formulated as a procedure that works by recursive descent on a formula and produces a set of equi-satisfiable set of clauses. A simple conversion algorithm introduces one fresh name for each sub-formula, and defines the name using the shape of the sub-formula. We here recall the basic idea by converting and-or formulas into CNF.

$$\begin{array}{lcll}
\mathit{cnf}(\varphi) & = & \textbf{let } (\ell, F) = \mathit{cnf}'(\varphi) \textbf{ in } \ell \wedge F & \\
\mathit{cnf}'(\ell) & = & (\ell, \mathit{true}) & \ell \text{ is a literal} \\
\mathit{cnf}'(\neg\varphi) & = & \textbf{let } (\ell, F) = \mathit{cnf}'(\varphi) \textbf{ in } (\neg\ell, F) & \\
\mathit{cnf}'(\varphi \wedge \psi) & = & \textbf{let } (\ell_1, F_1) = \mathit{cnf}'(\varphi),\ (\ell_2, F_2) = \mathit{cnf}'(\psi) \textbf{ in} & \\
 & & (p, F_1 \wedge F_2 \wedge (\neg\ell_1 \vee \neg\ell_2 \vee p) \wedge (\neg p \vee \ell_1) \wedge (\neg p \vee \ell_2)) & p \text{ is fresh} \\
\mathit{cnf}'(\varphi \vee \psi) & = & \textbf{let } (\ell_1, F_1) = \mathit{cnf}'(\varphi),\ (\ell_2, F_2) = \mathit{cnf}'(\psi) \textbf{ in} & \\
 & & (p, F_1 \wedge F_2 \wedge (\ell_1 \vee \ell_2 \vee \neg p) \wedge (p \vee \neg\ell_1) \wedge (p \vee \neg\ell_2)) & p \text{ is fresh}
\end{array}$$

More sophisticated CNF conversions that do not introduce fresh names for all sub-formulas exist [14]. They control the number of auxiliary literals and clauses introduced during clausification.

Z3 does not introduce auxiliary predicates during internalization of quantifier-free formulas. Instead, it re-uses the terms that are already used for representing the sub-formulas. Thus, instead of introducing a fresh variable p, in the CNF conversion of $\varphi \vee \psi$, we treat the term $\varphi \vee \psi$ as a *literal*. To clarify that a sub-formula plays the rôle of a literal below, we *quote* it. So the literal associated with $\varphi \vee \psi$ is $\lceil \varphi \vee \psi \rceil$. So the CNF conversion of $\varphi \vee \psi$ produces the pair:

$$(\lceil \varphi \vee \psi \rceil,\ F_1 \wedge F_2 \wedge (\ell_1 \vee \ell_2 \vee \neg\lceil \varphi \vee \psi \rceil) \wedge (\lceil \varphi \vee \psi \rceil \vee \neg\ell_1) \wedge (\lceil \varphi \vee \psi \rceil \vee \neg\ell_2))$$

It may appear that we need to justify the auxiliary clauses $(\ell_1 \vee \ell_2 \vee \neg\lceil \varphi \vee \psi \rceil)$, $(\lceil \varphi \vee \psi \rceil \vee \neg\ell_1)$, and $(\lceil \varphi \vee \psi \rceil \vee \neg\ell_2)$ by appealing to the equi-satisfiablility, but the justification for these clauses happens in fact to directly use equivalence. First note that it follows by induction on the clausification algorithm that ℓ_1 is equivalent to φ and ℓ_2 is equivalent to ψ. Each of the auxiliary clauses introduced during clausification is therefore justified as propositional tautologies. Only limited propositional reasoning is required to justify these. The proof terms corresponding to the auxiliary clauses are tagged as definitional axioms. For example, the axiom $\lceil \varphi \vee \psi \rceil \vee \neg\varphi$ is represented by the term $\mathit{def\_axiom}((\varphi \vee \psi) \vee \neg\varphi)$.

We introduced quotation here to clarify in which context logical connectives were to be used. Our implementation in Z3 does not use quotation at all.

3.3.2 Arithmetic

Implicit quotation is also used when introducing auxiliary variables for theories, such as the theory for linear arithmetic. Let us recall how the Simplex solver in Z3 works. Following [5], a theory solver for linear arithmetic, and integer linear arithmetic can be based on a Simplex Tableau of the form:

$$x_i \simeq \sum_{x_j \in \mathcal{N}} a_{ij} x_j \quad x_i \in \mathcal{B}, \tag{1}$$

where $\mathcal{B}$ and $\mathcal{N}$ denote the set of basic and nonbasic variables, respectively. The sets $\mathcal{B}$ and $\mathcal{N}$ are assumed disjoint, and each basic variable occurs in exactly one row. Thus, the values of basic variables are determined by the values of the non-basic variables. In addition to this tableau, the solver state stores upper and lower bounds l_i and u_i for every variable x_i and a mapping β that assigns a rational value $\beta(x_i)$ to every variable x_i. The bounds on nonbasic variables are always satisfied by β, so the following invariant is maintained by the tableau operations

$$\forall x_j \in \mathcal{N},\ \ l_j \leq \beta(x_j) \leq u_j. \tag{2}$$

A tableau is *satisfied* if the same inequalities hold for the basic variables:

$$\forall x_j \in \mathcal{B},\ \ l_j \leq \beta(x_j) \leq u_j. \tag{3}$$

Bounds constraints for basic variables are not necessarily satisfied by β, so for instance, it may be the case that $l_i > \beta(x_i)$ for some basic variable x_i, but pivoting steps can be used to fix bounds violations, or detect an unsatisfiable tableau.

Formally, Simplex-based solvers for linear arithmetic can interface with Boolean combinations of inequalities by introducing one slack variable and a tableau row for every (maximal) linear arithmetic sub-term in a formula. For example, it is by now common for SMT solvers to transform problems of the form:

$$[(x + y > 2) \wedge (2x - y < 2)] \vee [(f(z + x) \geq 5) \wedge z < 3] \tag{4}$$

to the following equi-satisfiable formula:

$$\begin{array}{rl} & [(s_1 > 2) \wedge (s_2 < 2)] \vee [(f(s_3) \geq 5) \wedge z < 3] \\ \wedge & s_1 \simeq x + y \wedge s_2 \simeq 2x - y \wedge s_3 \simeq z + x \end{array} \tag{5}$$

The definitions for the slack variables translate into rows in a Simplex tableau. If we represent s_1 by the quotation $\lceil x{+}y \rceil$, and s_2 by $\lceil 2x{-}y \rceil$, and s_3 by $\lceil z{+}x \rceil$, we observe directly that the additional equalities are tautologies and therefore have trivial justifications. Furthermore, all Simplex tableau operations, including pivoting, are equivalence preserving (pivoting replaces equals for equals, and divides rows by constants), so the justifications for the Simplex rows remains a matter of expanding quotations and checking equalities using linear arithmetic. The above observation can be used to reconstruct proofs from unsatisfiable tableaux in a direct manner. In contrast, [2] proposed an encoding of arithmetical constraints by introducing slack variables for potentially every arithmetical subterm (including potentially a slack variable for z in $z < 3$). The slack variables allowed for tracking explanations and extracting proofs from infeasible tableaux.

We now explain how proofs are extracted from unsatisfiable tableaux without modifying the translation phase. With a tableau row of the form (1) associate the suprema and infima of implied by the coefficients, namely, let a_r be the coefficients from the row and l_j and u_j be the lower and upper bounds that are asserted by the literals $x_j \leq u_j$ and $l_j \leq x_j$, then:

$$\mathsf{sup}(a_r) \ := \sum_{x_j \in \mathcal{N}^+} a_{rj} u_j + \sum_{x_j \in \mathcal{N}^-} a_{rj} l_j \tag{6}$$

$$\mathsf{inf}(a_r) \ := \sum_{x_j \in \mathcal{N}^+} a_{rj} l_j + \sum_{x_j \in \mathcal{N}^-} a_{rj} u_j \tag{7}$$

where $\mathcal{N}^- = \{x_j \mid a_{rj} < 0\}$, and $\mathcal{N}^+ = \{x_j \mid a_{rj} > 0\}$; and as usual, we set $\mathsf{sup}(a_r) = \infty$ if either some $x_j \in \mathcal{N}^+$, $u_j = \infty$, or for some $x_j \in \mathcal{N}^-$, $l_j = -\infty$.

Then an unsatisfiable tableau can be identified by an infeasible row r, where

$$u_r < \mathsf{inf}(a_r) \text{ and } x_r \le u_r \text{ is asserted, or } l_r > \mathsf{sup}(a_r) \text{ and } l_r \le x_r \text{ is asserted.} \tag{8}$$

Corresponding to an infeasible row, we can extract a theory conflict by accumulating the bounds that were used to derive a contradiction. So for example, in case of $u_r < \mathsf{inf}(x_r)$, the conflict clause is of the form

$$\neg(x_r \le u_r) \vee \bigvee_{x_j \in \mathcal{N}^+} \neg(l_j \le x_j) \vee \bigvee_{x_j \in \mathcal{N}^-} \neg(x_j \le u_j) \tag{9}$$

It is now simple to prove the conflict clause:

$$lemma\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x_r \simeq \sum_{x_j \in \mathcal{N}} a_{rj}x_j \qquad l_{r1} \le x_{r1},\ x_{r1} \in \mathcal{N}^+}{x_r \ge (\sum_{x_j \in \mathcal{N}} a_{rj}x_j) - a_{r1}x_{r1} + a_{r1}l_{r1}} \qquad x_{r2} \le u_{r2},\ x_{r2} \in \mathcal{N}^-}{x_r \ge (\sum_{x_j \in \mathcal{N}} a_{rj}x_j) - a_{r1}x_{r1} + a_{r1}l_{r1} - a_{r2}x_{r2} + a_{r2}u_{r2}} \qquad \dots}{\dots}}{x_r \ge \mathsf{inf}(a_r)} \qquad x_r \le u_r}{\bot}}{\neg(x_r \le u_r) \vee \bigvee_{x_j \in \mathcal{N}^+} \neg(l_j \le x_j) \vee \bigvee_{x_j \in \mathcal{N}^-} \neg(x_j \le u_j)\ (9)}$$

The left-most antecedent is a tautology in the theory of linear arithmetic, the other antecedents are hypotheses. The $lemma$ inference rule collects (see Section 3.4) the disjunction of the negated hypotheses used for deriving $\bot$. The intermediary inferences correspond to basic inequality propagation. Our implementation in Z3 only produces the theory lemma directly without listing the equality corresponding to the infeasible row.

3.4 Modular Proofs

A basic underlying principle for composing and building proofs in Z3 has been to support a modular architecture that works well with theory solvers that receive literal assignments from other solvers and produce contradictions or new literal assignments. The theory solvers should be able to produce independent and opaque explanations for their decisions.

Conceptually, each solver acts upon a set of hypotheses and produce a consequent. The basic proof-rules that support such an architecture can be summarized as: $hypothesis$, that allow introducing an assumption, $lemma$, that eliminates hypotheses, and $unit\_resolution$ that handles basic propagation. We say that a proof-term is *closed* when every path that ends with a hypothesis contains an application of rule *lemma*. If a term is not closed, it is *open*. To summarize, these core rules are:

$hypothesis(\varphi)$ Mark φ as a hypothesis. The resulting proof term is open.

$lemma(p, \neg\varphi_1 \vee \ldots \vee \neg\varphi_n)$ The proof term has one antecedent p such that $con(p) =$ *false*, but p is open with hypotheses $\varphi_1, \ldots, \varphi_n$. The resulting proof term is closed.

$unit\_resolution(p_0, p_1, \ldots, p_n, \psi_1 \vee \ldots \vee \psi_m)$ Where $con(p_0) = \varphi_1 \vee \ldots \vee \varphi_n \vee \psi_1 \vee \ldots \vee \psi_m$, $con(p_1) = \neg\varphi_1, \ldots con(p_n) = \neg\varphi_n$.

We will next describe how these rules integrate within a DPLL(T) architecture.

3.4.1 Proofs from DPLL(T)

The propositional inference engine in Z3 is based on a DPLL(T) architecture. We refer to [13] for an exposition on a basic introduction on DPLL(T) as a transition system. The main points we will use is that DPLL(T) maintains a state of the form $M \parallel F$ during search, where M is a partial assignment of the atomic predicates in the formula F. Furthermore, we assume F is in conjunctive normal form. The search keeps assigning atoms in M based on unit propagation, *theory* propagation (for example $x > 3$ implies that $x > 0$ by the theory of arithmetic), and guesses (also called decisions) until either it reaches a state where the assignment satisfies all clauses in F, or some clause in F contradicts the assignment in M.

The DPLL(T) proof search method lends itself naturally to producing resolution style proofs. Systems, such as zChaff, and a version of MiniSAT [10, 6, 21], produce proof logs based on logging the unit propagation steps as well as the conflict resolution steps. The resulting log suffices to produce a propositional resolution proof. This approach works even though the SAT solver can choose to restart or garbage collect learned conflict clauses that were produced during search.

The approach taken in Z3 bypasses logging, and instead builds proof objects during conflict resolution. With each clause we attach a proof. Clauses that were produced as part of the input have proofs that were produced from the previous steps. A clause that is produced during conflict resolution depends on some state of the partial model M. In particular, the learned clause is contradictory with some subset of the decision literals in M, either directly because the learned clause contains decision literals, or because the learned clause contains a literal that was obtained by propagation. Given a conflict clause $C : \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$, we build a proof term of the form $lemma(p, \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n)$, where p is constructed by examining the justifications for $\neg\ell_1, \ldots, \neg\ell_n$. If $\neg\ell_i$ is a decision literal, then the justification for $\neg\ell_i$ is a term $hypothesis(\neg\ell_i)$. If $\neg\ell_i$ was inferred by unit propagation (so there is a fact $\neg\ell_i$ in M, with justification $C \vee \neg\ell_i$), then it is proved using unit-resolution and the justification for the clause $C \vee \neg\ell_i$.

We see that this approach does not require logging resolution steps for every unit-propagation, but delays the analysis of which unit propagation steps are useful until conflict resolution. The approach also does not produce a resolution proof directly. It produces a natural deduction style proof with hypotheses.

Other propositional rules that are used during proof-reconstruction are:

$asserted(\varphi)$ The formula φ is a user-supplied assumption.

$goal(\varphi)$ The formula φ is a user-supplied goal. A goal is symmetric to *asserted*, but allows retaining the distinction between goals and assumptions in proof objects.

$mp(p, q, \varphi)$ Proof of φ by modus ponens. Assume that $con(p) = \psi$ and that $con(q)$ is either $\psi \rightarrow \varphi$ or $\psi \leftrightarrow \varphi$. The latter form is used extensively in the simplifier to apply equivalence-preserving simplification steps.

3.4.2 Congruence Proofs

In Z3, the congruence closure implementation is tightly integrated with the Boolean satisfiability core. It serves as a main hub for equality propagation. The efficient extraction of minimal justifications for congruence closure proofs has been studied extensively, [11]. We here summarize the proof objects that are extracted from the justifications.

The theory of equality can be captured by axioms for reflexivity, symmetry, transitivity, and substitutivity of equality. We encode these axioms as inference rules, and furthermore only specify that these inference rules apply for any binary relation that is reflexive, symmetric, transitive, and/or reflexive-monotone. We use the terminology, reflexive-monotone, for relations that are reflexive and monotone in a given function symbol f. In particular, the relation $\sim$ (from Section 2.1) is also an equivalence relation, and reflexive-monotone over conjunction and disjunction. So the rules are:

$\mathit{refl}(R(t,t))$ A proof for $R(t,t)$, where R is a reflexive relation.

$\mathit{symm}(p, R(t,s))$ A proof of $R(t,s)$, where R is a symmetric relation, and $\mathit{con}(p) = R(s,t)$.

$\mathit{trans}(p, q, R(t,s))$ A proof of $R(t,s)$, where R is transitive, and $\mathit{con}(p) = R(t,u)$ and $\mathit{con}(q) = R(u,s)$.

$\mathit{monotonicity}(p_1, .., p_n, R(f(t_1, .., t_n), f(s_1, .., s_n)))$ A proof of $R(f(t_1, .., t_n), f(s_1, .., s_n))$, where $\mathit{con}(p_1) = R(t_1, s_1)$, $\ldots$, $\mathit{con}(p_n) = R(t_n, s_n)$, and R is reflexive and monotone in f. The antecedent p_i can be suppressed if $t_i = s_i$. That is, reflexivity proofs are suppressed to save space.

Our congruence closure core maintains a congruence table. A congruence table enables propagation of equalities over function symbols, so that for example if $f(s,t)$ is a term, and s' is equal to s, then when creating $f(s',t)$ it is detected that the potentially new term is in the same congruence class as $f(s,t)$. The implementation also treats equality as a function, and every equality is also inserted to the congruence table. This makes detecting implied dis-equalities simple: given two terms s and t, search the congruence table for an existing entry for $s \simeq t$. If the table contains such a literal, then check if the literal is assigned to *false*. To make this use of the congruence table effective it understands commutative operations, such as equality. This implicit use of commutativity gets reflected in the generated proof terms, and we include a special rule for commutative functions. It can be instantiated for equalities.

$\mathit{comm}(f(s,t) \simeq f(t,s))$, $\mathit{comm}(f(s,t) \leftrightarrow f(t,s))$ where f is a commutative function (relation).

3.4.3 Theory lemmas

In the DPLL(T) architecture, decision procedures for a theory T identify sets of asserted T-inconsistent literals. Dually, the disjunction of the negated literals are T-tautologies. Consequently, proof terms created by theories can be summarized using a single form, here called Theory lemmas.

$\mathit{th\_lemma}(p_1, \ldots, p_n, \varphi)$ Generic proof rule for theory lemmas. The formula $\mathit{con}(p_1) \wedge \ldots \wedge \mathit{con}(p_n) \rightarrow \varphi$ should be a T-tautology.

3.5 Applications

Z3 with proofs is still under development and has not been released yet. An obvious current application is that proofs offer a simple litmus test on the implementation for soundness bugs. We integrate a simple, but partial (it does not check T-lemmas) proof-checker in Z3 for this purpose. A much more effective strategy for debugging bugs in theory solvers has been to dump the T-lemmas as they are produced. Similar to [20], we can then apply an independent solver (namely, our previous version of Z3) on the T-lemmas. We found this approach very effective in debugging optimizations that turned out to be unsound.

We also have a facility for displaying proof-terms, but the proof term visualization very easily becomes too large to be of any use.

In future applications, we envision applications of proofs in Z3, such as: Proof-mining [17]; can proofs be mined for strategies that are helpful for speeding up proofs for a class of problems? Interpolation. Proof visualization. Finally, in the context of Isabelle/HOL, it has been suggested [8] to translate HOL formulas (which use polymorphism), into first-order untyped formulas. A potentially unsound translation is then run through first-order provers, but the produced proofs (currently apparently only Prover9), can be checked for whether they use inferences that contradict the types.

3.6 The overhead of enabling proofs

We benchmarked proof generation on a few selected, but non-trivial examples from SMT-LIB. The samples show a memory overhead of between 3x-40x, and corresponding slowdowns of 1.1x to 3x.

Benchmark	Without Proofs		With Proofs	
NEQ016_size7	26.01 secs	9.1 MB	39.84 secs	426 MB
PEQ010_size8	6.65 secs	9.3 MB	7.63 secs	40 MB
fischer6-mutex-14	7.01 secs	14 MB	16.38 secs	142 MB
cache.inv16	15.99 secs	11 MB	24.61 secs	159 MB
xs_20_40	2.2 secs	5.8 MB	2.70 secs	15.5 MB

4 Models

We will very briefly summarize the model generation features in Z3. More material is available on-line on `http://research.microsoft.com/projects/z3/models.html`.

Z3 has the ability to produce models as part of its output. Models assign values to the constants in the input and generate finite or partial function graphs for predicates and function symbols.

A model comprises of a set of partitions, each partition is printed as: `*`k, where k is a non-negative number. Each partition is associated with a set of constants from the input, and is associated with a concrete value. A concrete value can be either a Boolean (*true* or *false*), a numeral (with type Int, Real, or BitVec[n]), an array (represented as a finite map), a tuple (represented by a constructor and sequence of values for the fields), or an an uninterpreted ur-element. Ur-elements are internally represented as natural number numerals, but with an uninterpreted type.

By default Z3, produces a full (and compact) interpretation for free functions. There is an option to force Z3 to not assign interpretations to functions when their values don't influence the truth assignment to the formula.

Z3 version 2 integrates a superposition theorem prover [4]. When it is able to finitely saturate the set of non-ground input clauses, it can also report that the non-ground formula that was provided is satisfiable, but there are no other mechanisms for extracting additional information from the saturated set of clauses.

4.1 Applications

Models are by now used in a number of Z3 clients. The main clients that use models are the program exploration and test-case generation tools Pex and SAGE (we refer to [3] for all pointers). They extract symbolic path conditions by monitoring program executions and use Z3 to find alternate inputs that can guide the next execution into a different branch. Models are also used for improved debugging feedback from Spec# and for iterative counter-example guided refinement in the context of bounded model checking of model programs.

We also believe that the availability of models can in future applications play a useful role in the context of model-based quantifier instantiation and integrating external decision procedures with Z3.

5 Conclusions

We presented the proof and model generation facilities in Z3. Models have already shown particular usefulness in the context of SMT applications. Proofs will be available in Z3 v2, and we hope, in light of this introduction, that users will be able to find useful applications of the feature.

References

[1] Aaron Stump and Duckki Oe. Towards an SMT Proof Format. In *6'th International Workshop on SMT*, 2008.

[2] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In Ramakrishnan and Rehof [15], pages 397–412.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In Ramakrishnan and Rehof [15].

[4] Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + Saturation. In *IJCAR'08*, 2008.

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

[6] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[7] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[8] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[9] Michal Moskal. Rocket-Fast Proof Checking for SMT Solvers. In Ramakrishnan and Rehof [15], pages 486–500.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[11] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[12] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In Robinson and Voronkov [16], pages 371–443.

[13] Roberto Niewenhuis, 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.

[14] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [16], pages 335–367.

[15] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[16] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[17] S. Schulz. Learning Search Control Knowledge for Equational Theorem Proving. In F. Baader, G. Brewka, and T. Eiter, editors, *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 320–334. Springer, 2001.

[18] Aaron Stump, Clark W. Barrett, and David L. Dill. Producing proofs from an arithmetic decision procedure in elliptical lf. *Electr. Notes Theor. Comput. Sci.*, 70(2), 2002.

[19] Aaron Stump and David L. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2002.

[20] Geoff Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[21] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, July 2007.

[22] Yeting Ge and Clark Barrett. Proof Translation and SMT-LIB Benchmark Certification: A Preliminary Report. In *6'th International Workshop on SMT*, 2008.