Engineering Theories with Z3*

Nikolaj Bjørner

Microsoft Research
nbjorner@microsoft.com

Abstract. Modern Satisfiability Modulo Theories (SMT) solvers are fundamental to many program analysis, verification, design and testing tools. They are a good fit for the domain of software and hardware engineering because they support many domains that are commonly used by the tools. The meaning of domains are captured by theories that can be axiomatized or supported by efficient *theory solvers*. Nevertheless, not all domains are handled by all solvers and many domains and theories will never be native to any solver. We here explore different theories that extend Microsoft Research's SMT solver Z3's basic support. Some can be directly encoded or axiomatized, others make use of user theory plug-ins. Plug-ins are a powerful way for tools to supply their custom domains.

1 Introduction

This paper surveys a selection of theories that have appeared in applications of Z3 [7] and also in recent literature on automated deduction. In each case we show how the theories can be supported using either existing built-in theories in Z3, or by adding a custom decision procedure, or calling Z3 as a black box and adding axioms between each call. The theme is not new. On the contrary, it is very central to research on either encoding (reducing) theories into a simpler basis or developing special solvers for theories. Propositional logic is the most basic such basis e.g., [14]. In the context of SMT (Satisfiability Modulo Theories), the basis is much richer. It comes with built-in support for the theory of equality, uninterpreted functions, arithmetic, arrays, bit-vectors, and even first-order quantification. The problem space is rich, and new applications that require new solutions keep appearing. We don't offer a silver bullet solution, but the "exercise" of examining different applications may give ideas how to tackle new domains.

Z3 contains an interface for plugging in custom theory solvers. We exemplify this interface on two theories: MaxSMT (Section 3) and partial orders (Section 4). This interface is powerful, but also requires thoughtful interfacing. To date it has been used in a few projects that we are aware of [18, 2, 16]. Some of our own work can also be seen as an instance of a theory solver. The quantifier-elimination procedures for linear arithmetic and algebraic data-types available in Z3 acts as a special decision procedure [3]. The OpenSMT solver also supports

an interface for pluggable theories [5]. We feel that the potential is much bigger an we conclude with some speculation where the pluggable interface could be used elsewhere.

Z3 also allows interfacing theories in simpler ways. The simplest is by encoding and Section 5 discuss a simple theory with two encodings. Something between encoding and a user theory, is by calling Z3 repeatedly. Whenever Z3 returns a satisfiable state, then add new axioms that are not satisfied by the current candidate model for the existing formulas. Section 6 discusses how HOL can be solved using this method.

The case studies discussed in this paper are available as F# code samples.

2 SMT, DPLL(T), and Z3

2.1 SMT

We will not survey SMT here, but refer the reader to [8] for an introduction.

2.2 DPLL(T)

Modern SMT solvers are mostly based on the DPLL(T) architecture. In this context, an efficient propositional SAT solver is used to produce a truth assignment to the atomic sub-formulas of the current goal. Let us use M to refer to a partial assignment. It can be represented as a stack of literals $\ell_1, \ell_2, \ldots, \ell_n$. The partial assignment is updated by adding new literals to the stack to indicate their values, and by shrinking the stack. We use F for the current goal. The DPLL(T) architecture uses two main methods for interacting with a theory solver.

T-Propagate Given a state $M \parallel F$, such that ℓ or $\neg\ell$ occurs in F, ℓ is unassigned in M, $\overline{C} \subseteq M$ (the negation of the literals in C are already assigned in M), and $T \vdash C \vee \ell$, then ℓ must be *true* under the current assignment M. It is then sound to propagate ℓ.

External theory solvers in Z3 can force this propagation by asserting the clause $(C \vee \ell)$. Then ℓ gets assigned by propositional propagation. However, the asserted clause has no value if ℓ does not participate in a conflict. So the default behavior in Z3 is to garbage collect the asserted clause on backtracking.

T-Conflict Given a state $M \parallel F$ such that $\overline{C} \subseteq M$, $T \models C$. That is, there is a subset $\overline{C}$ of the literals in M that are inconsistent from the point of view of T, or dually the clause C is T-valid, then assert the valid clause C. The new clause is in conflict with the current assignment M, because all literals in C are *false* under M. The propositional engine detects the resulting conflict and causes backtracking.

The clause C may also be useless beyond serving the role of signaling the conflict. It is therefore also by default garbage collected during Z3's backtracking.

2.3 Z3's Theory Solver API

Z3 exposes a programmatic API for interfacing with the theory solver. It includes hooks for user theories to add callbacks so that they can implement the effect of T-Propagate and T-Conflict. As we saw, the effect of the rules is communicated by asserting a new clause to the current state. The corresponding method is called AssertTheoryAxiom over the .NET API. On the other hand, the state changes to the partial model M are exposed to the user solver using callbacks. When a literal is added to M (is assigned to either *true* or *false*), then a callback (called NewAssignment(*atom*,*truth_value*)) is invoked with the underlying atomic formula and the truth value it is assigned to. There are other specialized callbacks when new equalities and dis-equalities are discovered. Equalities and dis-equalities don't have to correspond to existing atoms. Finally, a callback (called FinalCheck in .NET) is invoked when the current assignment fully satisfies the current formula F from the point of view of the built-in theories in Z3. The user theory solver can inspect its own state and compare the assignment it learned from NewAssignment to determine if the resulting assignment is satisfiable. When the solver performs a new case split or backtracks through states it calls into the theory solver with callbacks Push and Pop. Any side-effects made in a user-theory inside the scope of a Push need to be undone when receiving a matching call to Pop. For example, if a user-theory performs an update $c \leftarrow c + w_i$, where w_i is a constant, then it can undo the effect of the operation by executing $c \leftarrow c - w_i$ during a Pop.

3 Weighted MaxSMT

Weighted MaxSMT is the following problem. Given a set of numeric *weights* $w_1, \ldots, w_n$ and formulas $F_0, F_1, \ldots, F_n$, find the subset $I \subseteq \{1, \ldots, n\}$ such that

1. $F_0 \wedge \bigwedge_{i \notin I} F_i$ is satisfiable.
2. The *cost*: $\Sigma_{i \in I} w_i$ is minimized.

In other words, the weight w_i encodes the penalty for a formula F_i to not be included in a satisfying assignment. The paper [15] develops a theory solver for weighted MaxSMT. An important point is that the theory evolves as search progresses: once a satisfiable state is reached with a given cost c, then assignments that meet or exceed c are useless.

According to [15], weighted MaxSMT can be encoded in Z3 in the following way: Initially we assert F_0 and $F_i \vee p_i$ for each i, where p_i is a fresh propositional variable. We also maintain a cost c that is initialized to 0, and a *min_cost* that is set to *nil*. Then, repeat the following steps until the asserted formulas are unsatisfiable:

1. When some p_i is assigned to *true*, then update $c \leftarrow c + w_i$.
2. If $nil \neq min\_cost \leq c$, then block the current state by calling AssertTheoryAxiom($\bigvee\{\neg p_i \mid p_i$ is assigned to *true*$\}$).

3. When receiving the FinalCheck callback, it must be the case that $c < min\_cost$ or min_cost is *nil*. So it is safe to set $min\_cost \leftarrow c$. To block this current cost call AssertTheoryAxiom($\bigvee\{\neg p_i \mid p_i$ is assigned to *true*$\}$).

It was tempting to make fuller use of theory support in Z3 for MaxSMT. We also tried an encoding that used extra variables $v_1, \ldots, v_n$ and axioms $p_i \implies v_i = w_i$, $\neg p_i \implies v_i = 0$, $0 \leq v_i$ for each $i = 1, \ldots, n$ and $\Sigma_i v_i < B$. The idea would be to add assertions $B \leq c$ every time a new satisfying state with cost c is encountered. This encoding was counter-productive on the benchmarks used in [15], it taxes Z3's arithmetic solver in contrast to relatively cheap propagation using the blocking propositional clauses.

It can be highly domain dependent whether a particular solution applies. An example of constraints where the proposed MaxSMT solver performs poorly comes from constraints used to tune parameters for the Vampire [12] theorem prover. There, a formula $F_0[v_1, \ldots, v_n]$ is asserted and the goal is to minimize $\Sigma_i v_i$. The domain of the variables is bounded and the so-far best encoding appears to be to use bit-vectors for the variables. We can convert the problem to MaxSMT by adding the following soft clauses: $v_i[k] = 0$ with weight 2^k for each variable v_i of bit-width N and $0 \leq k < N$. Nevertheless, we found that this encoding is inferior to the best technique known so far: a binary search over constraints of the form $\Sigma_i v_i > c$, where c is a candidate lower bound.

4 Theories for partial orders and class inheritance

A partial order is a binary relation that is reflexive, anti-symmetric, and transitive. In other words, $\preceq$ is a partial order if for every x, y, z:

$$x \preceq x, \quad x \preceq y \wedge y \preceq x \implies x = y, \quad x \preceq y \wedge y \preceq z \implies x \preceq z$$

When there are no other non-ground properties of $\preceq$, it is relatively straightforward to support the theory using axioms that get instantiated fully during search. Unfortunately, the theory is *expensive*. When n is the number of terms in the goal that occur in either side of $\preceq$, the axiom for transitivity causes up to $O(n^3)$ clauses and generates up to $O(n^2)$ instantiations of $\preceq$. The (quantifier-free) theory of partial orders can be solved using graph search procedures. Let us illustrate two theory solvers in the context of partial orders.

4.1 A basic solver for partial orders

We present a basic decision procedure for the theory of partial orders. It maintains a directed graph $\mathcal{D}$ and a set $\mathcal{N}$ of pairs of terms. They are both initially empty. Assert F, the original formula to satisfy and check for satisfiability with the following theory solver actions:

1. When $t \preceq t'$ is asserted to *true*, then add the edge $t \rightarrow t'$ to $\mathcal{D}$.
2. When $t = t'$ is asserted, then add edges $t \rightarrow t' \rightarrow t$ to $\mathcal{D}$.

3. If $\mathcal{D}$ contains a cycle with edges $t_1 \to t_2 \to t_3 \to ..t_n \to t_1$, for terms that are not yet asserted equal, then call

$$\mathsf{AssertTheoryAxiom}(t_1 \preceq t_2 \preceq t_3 \ldots t_n \preceq t_1 \to \bigwedge_{i=1}^{n-1} t_i = t_{i+1})$$

4. When $t \preceq t'$ is asserted to *false*, then add the pair (t, t') to $\mathcal{N}$.
5. If for some pair (t, t') in $\mathcal{N}$ there is a path in $\mathcal{D}$ from t to t' (the path can be empty and $t = t'$), then assert

$$\mathsf{AssertTheoryAxiom}(t \preceq t_1 \preceq t_2 \ldots t_n \preceq t' \to t \preceq t')$$

Correctness of the algorithm is straight-forward: The graph $\mathcal{D}$ is a model: every term in strongly connected components are forced equal, and every constraint $t \not\preceq t'$ is checked. It is critical that the algorithm has access to the current equalities between terms and it takes part of equality propagation as well.

A basic implementation of the corresponding solver is to defer all processing to FinalCheck. Tarjan's ubiquitous linear time algorithm for finding strongly connected components in a graph will identify implied equalities, and each pair in $\mathcal{N}$ can be checked in time $|\mathcal{D}|$.

4.2 Sub-typing Closure

The object inheritance hierarchy of classes in object oriented programs forms a partial order. A special class of partial order constraints are relevant in this context, and [17] develops a specialized decision procedure. In the context of object inheritance we can assume there is a fixed set of constants $cls_1, \ldots, cls_n$ that are (1) all distinct and (2) covers the universe of types that are used in the query. The type hierarchy among $cls_1, \ldots, cls_n$ is fixed once and the queries are Boolean formulas over atoms of the form $x = cls_i$ and $x \preceq cls_i$, where x is a variable (it is equal to one of $cls_1, \ldots, cls_n$, but the concrete value is not known yet.

The theory can be handled using a specialized solver that tracks satisfiability of assignments to the atoms: For each variable x, initialize the set of candidates $cand(x)$ to $\{cls_1, \ldots, cls_n\}$, and dependencies $dep(x)$ to $\emptyset$.

1. The state is updated upon asserting a literal ℓ as follows:
   (a) $x = cls_i$: set $cand(x) \leftarrow cand(x) \cap \{cls_i\}$.
   (b) $x \neq cls_i$: set $cand(x) \leftarrow cand(x) \setminus \{cls_i\}$.
   (c) $x \preceq cls_i$: intersect $cand(x)$ with the descendants of cls_i.
   (d) $x \not\preceq cls_i$: subtract the descendants for cls_i from $cand(x)$.
2. The asserted literal ℓ is also added to $dep(x)$.
3. The state is unsatisfiable if $cand(x) = \emptyset$. To block it, call:

$$\mathsf{AssertTheoryAxiom}(\bigvee\{\neg\ell \mid \ell \in dep(x)\})$$

The interesting problem is implementing the updates to $cand(x)$ efficiently. The assumption that the type hierarchy is fixed can be exploited. The Type Slicing [10] structure was developed in the context of fast dispatch tables for object oriented programs, and it was used in [17] for making the updates to $cand(x)$ efficient. The data-structure represents a partial order (directed acyclic graph) using a set of colored nodes that are ordered. The data-structure satisfies the following condition: For every node n and color c, the set of descendants of n of color c are contiguous with respect to the ordering. The contiguity requirement allows to represent descendants using the first and last element only of the interval. We will not review this data-structure and the methods for building it here, but note that the sketched solver integration with Z3 allows writing only the theory solver, while efficient handling of Boolean case splitting comes for free.

Remark 1. When detecting a conflict we suggested to include the negation of all literals from $dep(x)$ in the asserted theory axiom. The resulting axiom may have redundancies. For example if we assert $x = \mathsf{string}$ followed by $x \preceq \mathsf{System.Object}$ followed by $x = \mathsf{bool}$, we obtain a conflict by just producing the clause $x \neq \mathsf{string} \vee x \neq \mathsf{bool}$. The constraint $x \preceq \mathsf{System.Object}$ is redundant. A simple method is to minimize the conflicting dependencies for x by temporarily removing each literal from $dep(x)$ and check if there is still a conflict. Generating minimal conflicts is important for efficient search.

5 A Theory of Object Graphs

There are many cases where a new theory can already be encoded using existing built-in theories. There is then no need for special purpose procedures. Still there may not be a unique way to encode these theories. We here give an example of this situation.

The theory of object graphs uses elements from the theory of algebraic and co-algebraic data-types, yet it is not possible to directly use one or the other. The theory is also non-extensional. The theory of object graphs occurs naturally in the context of Pex [11]. Pex is a state-of-the-art tool for unit-test case generation. It applies to typed .NET code. Let us here consider the following program fragment:

```
class O {

   public readonly D d;
   public readonly O left;
   public O right;

   public O(D data,
            O left,
            O right) {
     this.data = data;
     this.left = left;
     this.right = right;
   }
}
```

```
void f(O n0) {

  Assert (n0 == null ||
          n0.left != n0);

  O n1 = new O(1,null,null);
  O n2 = new O(2,n1,null);
  O n3 = new O(2,n1,null);

  Assert(n2 != n3);

  n1.right = n2;
  n2.right = n1;
...
```

Program 5.1:

Objects of type O are created using a constructor that we also call O. Each allocation creates a different object (the default equality method is reference equality), so in the program `n2` is different from `n3`. We can use a heap, here called H, to track the state of objects. So access and updates to objects is done through the heap. The signature that is relevant for O is:

sorts:	O,
constructors:	$null : O, O : H \times D \times O \times O \to H \times O$,
accessors:	$data : H \times O \to D, left : H \times O \to O, right : H \times O \to O$,
modifiers:	$update\_right : H \times O \times O \to H$

The sort is O and there is a distinguished constant *null*. There are three accessors, the *data* accessor retrieves a data field from objects of type O, and *left* and *right* access left and right children. The read-only declared attributes of O cannot be updated, so there is only a single modifier for the *right* attribute.

The theory of O is characterized as follows:

$$
\begin{aligned}
&(h', o) = O(h, d, l, r) \implies o \neq null \\
&(h', o) = O(h, d, l, r) \implies data(h', o) = d \\
&(h', o) = O(h, d, l, r) \implies left(h', o) = l \\
&(h', o) = O(h, d, l, r) \implies right(h', o) = r \\
&left(null) = right(null) = null \\
&h' = update\_right(h, o, r) \wedge o \neq null \implies right(h', o) = r \\
&h' = update\_right(h, o, r) \wedge o' \neq o \implies right(h', o') = right(h, o') \\
&h' = update\_right(h, o, r) \implies left(h', o') = left(h, o') \\
&h' = update\_right(h, o, r) \implies data(h', o') = data(h, o')
\end{aligned}
$$

The read-only field constrains what objects are possible in a valid heap state. In particular all formulas of the form

$$o \neq \mathit{null} \implies \mathit{left}(h_1, \mathit{left}(h_2, \mathit{left}(\ldots \mathit{left}(h_n, o)))) \neq o \tag{1}$$

are valid. The restriction is similar to the occurs check (well-foundedness) of recursive data-types. On the other hand, the attributes following paths using *right* need not be well-founded.

The question we will now address is: How can we equip a decision procedure for reasoning about ground formulas over the theory of O?

5.1 An Encoding using Arrays

A direct encoding of objects is to use one array per field. To enforce well-foundedness of left-access (see (1)) one can use a time-stamp. We use $O \Rightarrow D$ for the sort of arrays that map O to D, and encode the sort O as the set N of natural numbers. The sort H is a tuple with one array for *data*, other arrays for *left* and *right*, and finally a clock that we will increment when allocating new objects.

$$\begin{aligned} O &= N \\ H &= \langle \mathit{data} : O \Rightarrow D, \mathit{left} : O \Rightarrow O, \mathit{right} : O \Rightarrow O, \mathit{clock} : N \rangle \end{aligned}$$

The constant *null* is set to 0 and object allocation modifies the arrays maintained in H. The initial heap h_0 uses the value 0 for *clock*, such that allocated objects are different from *null*.

$$\begin{aligned} \mathit{null} &= 0 \\ \mathit{left}_0 &= \mathit{store}(\mathit{left}_0, \mathit{null}, \mathit{null}) \\ \mathit{right}_0 &= \mathit{store}(\mathit{right}_0, \mathit{null}, \mathit{null}) \\ h_0 &= \langle \mathit{data}, \mathit{left}_0, \mathit{right}_0, 0 \rangle \\ O(\langle \mathit{data}, \mathit{left}, \mathit{right}, \mathit{clock} \rangle, d, l, r) &= \begin{pmatrix} \mathit{let}\ o = \mathit{clock} + 1 \\ (\langle \mathit{store}(\mathit{data}, o, d), \mathit{store}(\mathit{left}, o, l), \\ \mathit{store}(\mathit{right}, o, r), \mathit{clock} + 1 \rangle, o) \end{pmatrix} \\ \mathit{data}(\langle \mathit{data}, \mathit{left}, \mathit{right}, \mathit{clock} \rangle, o) &= \mathit{select}(\mathit{data}, o) \\ \mathit{left}(\langle \mathit{data}, \mathit{left}, \mathit{right}, \mathit{clock} \rangle, o) &= \mathit{select}(\mathit{left}, o) \\ \mathit{right}(\langle \mathit{data}, \mathit{left}, \mathit{right}, \mathit{clock} \rangle, o) &= \mathit{select}(\mathit{right}, o) \\ \mathit{update\_right}(\langle \mathit{data}, \mathit{left}, \mathit{right}, \mathit{clock} \rangle, o, r) &= \langle \mathit{data}, \mathit{left}, \mathit{store}(\mathit{right}, o, r), \mathit{clock} \rangle \end{aligned}$$

To enforce well-foundedness in models produced by Z3 it suffices to enforce that the time-stamp (here it is the same as the natural number used to identify objects) on non-null objects is smaller on their left children. It suffices to assert

an axiom that gets instantiated for every use of $left(h, o)$[1]

$$\forall h : H, o : O \;.\; o \neq null \Longrightarrow 0 \leq left(h, o) < o \;.$$

5.2 An Encoding using Recursive Data-types and Arrays

Another option is to encode the read-only fields using the theory of algebraic data-types. We use a unique identifier field id to make sure allocated objects are distinct.

$$\begin{aligned} O &= null \mid O(id : N, data : D, left : O) \\ H &= \langle right : O \Rightarrow O, clock : N \rangle \end{aligned}$$

$$\begin{aligned} right_0 &= store(right_0, null, null) \\ h_0 &= \langle right_0, 0 \rangle \\ O(\langle right, clock \rangle, d, l, r) &= \begin{pmatrix} let\ clock' = clock + 1 \\ let\ o = O(clock', d, l) \\ (\langle store(right, o, r), clock' \rangle, o) \end{pmatrix} \\ data(h, O(id, d, l)) &= d \\ left(h, O(id, d, l)) &= l \\ left(h, null) &= null \\ right(\langle right, clock \rangle, o) &= select(right, o) \\ update\_right(\langle right, clock \rangle, o, r) &= \langle store(right, o, r), clock \rangle \end{aligned}$$

5.3 Not All Encodings are Equal

The advantage of using the built-in algebraic data-types becomes highly visible when the heap gets updated multiple times. For example, in one test we created 1000 objects and then verified that the left child of the first object remained unchanged after the 1000 updates. It takes Z3 18 seconds to instantiate 600,000+ array axioms and establish the equality using the array-based encoding. The second encoding can prove the same theorem in a small fraction of a second. Establishing (1) requires also about 18 seconds and 232,582 quantifier instantiations using the first encoding, and is establish instantaneously using the second encoding.

[1] The mechanism for achieving this in Z3 is to annotate quantified formulas using this term as a *pattern*

6 HOL

Sattalax [4] is a theorem prover for Church's Higher-Order Logic (HOL) [1] that is based on simple type theory with Hilbert's choice operator. It won the CASC division for higher-order logic in 2011. The main idea in Sattalax is to reduce problems in HOL to a sequence of SAT problems. Sattalax uses the MiniSAT SAT solver. This, apparent unsophisticated method, has an edge over current competing tools thanks to the highly tuned SAT solver MiniSAT, and a judicious combination of strategies in Sattalax. The Sattalax reduction uses several components: It searches for quantifier instances for quantified formulas. It then encodes satisfiability of quantifier-free formulas into propositional logic. The purpose of this section is very straight-forward. It is to show how to leverage an SMT solver for handling the encoding of ground formulas into propositional logic. The other much more profound challenge remains, and we don't address it here: sophistication and tuning for finding useful quantifier instantiations.

There is a set of variables $\mathcal{V}$ with elements $x, y, z, \ldots$. The theory HOL is based on simply typed λ calculus. It includes a special sort o of propositions and i of individuals. Types are of the form:

$$\sigma ::= i \mid o \qquad \tau ::= \sigma \mid \tau \to \tau$$

Furthermore, we use the notation $\overline{\tau}$ as a shorthand for $\tau_1, \ldots, \tau_n$ and $\overline{\tau} \to \sigma$ as a shorthand for $\tau_1 \to \ldots \to \tau_n \to \sigma$. Terms are of the form:

$$M, N ::= \lambda x : \tau \;.\; M \mid (M\ N) \mid x$$

We assume also a fixed set of interpreted constants:

$$\begin{array}{l} \mathit{false} : o, \quad \Longrightarrow : o \to o \to o, \\ \epsilon : (\tau \to o) \to \tau, \quad \forall : (\tau \to o) \to o, \quad = : \tau \to \tau \to o \qquad \text{for each type } \tau \end{array}$$

As usual, terms are assumed simply and well-typed: $(M\ N)$ can only be formed if M has type $\tau \to \tau'$ and N has type τ. We write M^τ for a term M with type τ (under a type environment Γ). Simply typed terms are strongly normalizing, so they admit $\beta\eta$ normal forms that we denote $M\downarrow$. Equality under α-renaming can be dealt with by using de-Bruijn indices. HOL is generally highly incomplete (it can encode Peano arithmetic) but it is complete under Henkin [13] semantics. Under the Henkin term-based semantics the set of values in every type τ comprises of the all the closed terms of type τ. This set is non-empty for every τ because we can always include $\epsilon(\lambda x : \tau.\mathit{false})$. The interpreted constants are characterized by

$$(\forall\ (\lambda x : \tau\ .\neg(M\ x)))\ \vee\ (M\ (\epsilon\ M)) \qquad \text{for every } M : \tau \to o \tag{2}$$

$$M = N \Leftrightarrow (\forall \lambda x : \tau\ .\ (M\ x) = (N\ x)) \qquad \text{for every } M, N : \tau \to \tau' \tag{3}$$

$$(\forall\ M) \Longrightarrow (M\ N) \qquad \text{for every } M : \tau \to o, N : \tau \tag{4}$$

together with the usual congruence properties of equality and the Boolean connectives $\Longrightarrow$ and *false* (and the definitions for derived abbreviations $\neg, \Leftrightarrow, \vee$, and

$\wedge$). Furthermore, $M{\downarrow} = M$ for every M. The main idea of Sattalax is to saturate a goal F under these properties. Since Sattalax is based on a SAT solver it also has to saturate with respect to the theory of equality. The main point made here is that this part of the theory propagation is already taken care of by SMT solvers that provide ground equality reasoning as a built-in feature. Saturation causes the properties to be instantiated by every constructable terms M, N. Two challenges arise, the first is to find a way to enumerate all constructable terms, the second is to enumerate the terms in an order that is useful for finding short proofs. In general, one must fairly enumerate every type τ and every term of type τ. We sketch a construction of sets of terms with free variables from the typing context Γ and of type τ as the set $\mathcal{T}[\Gamma;\tau]$. It is the least fixed-point under the membership constraints:

$$\begin{array}{lll} (\lambda x : \tau \ .\ M) \in \mathcal{T}[\Gamma;\tau \to \tau'] & \text{if} & M \in \mathcal{T}[\Gamma, x:\tau;\tau'] \\ (x\ M_1\ \ldots\ M_k) \in \mathcal{T}[\Gamma;\sigma] & \text{if} & (x : \overline{\tau} \to \sigma) \in \Gamma,\ M_i \in \mathcal{T}[\Gamma;\tau_i] \end{array}$$

The constructed terms are in $\beta\eta$ long normal form. We here assume that Γ is pre-populated with the constants *false* and $\Longrightarrow$ and for every type τ a corresponding instance of $\forall, \epsilon, =$. A useful approach for enumerating the terms is to fix a depth towards the number of times one is willing to use either of the saturation rules above and then enumerate all terms and types up to the fixed depth.

6.1 Leveraging theories

The translation of λ-terms into first-order terms can exploit the support for equality and propositional logic that already exists in the context of SMT solvers. We give the translation function $[\![\_]\!]$ to the right. It creates quoted terms $\lceil M \rceil$ for λ-terms that don't correspond to Z3-expressible terms. The quoted terms are treated as uninterpreted constants from Z3's point of view. The theory of *extensional* arrays furthermore lets us enforce that application is extensional without having to expand axioms for extensionality ourselves. In other words, the function select satisfies $(\forall x : \tau \ .\ \mathsf{select}(M, x) = \mathsf{select}(N, x)) \Longrightarrow M = N$. We can therefore replace (3) with only the left-to-right implication.

$$\begin{aligned} [\![(\forall\ M)]\!] &= \lceil (\forall\ M) \rceil \\ [\![(\epsilon\ M)]\!] &= \lceil (\epsilon\ M) \rceil \\ [\![M \Longrightarrow N]\!] &= [\![M]\!] \Longrightarrow [\![N]\!] \\ [\![M = N]\!] &= [\![M]\!] = [\![N]\!] \\ [\![(M\ N)]\!] &= \mathsf{select}([\![M]\!], [\![N]\!]) \\ [\![\lambda x : \tau \ .\ M]\!] &= \lceil \lambda x : \tau \ .\ M \rceil \\ [\![f]\!] &= f \quad \text{for constant } f \end{aligned}$$

We are now ready to outline the basic saturation loop for HOL. Initialize the depth $d \leftarrow 0$. Assert $[\![F\downarrow]\!]$. Then repeatedly apply the following steps until $[\![F\downarrow]\!]$ is ground unsatisfiable:

1. F contains the sub-term $\lceil (\epsilon\ M) \rceil$, then add $[\![(2)\downarrow]\!]$ to F.
2. F contains the sub-term $[\![M^{\tau\to\tau'} = N]\!]$, then add $[\![(3)\downarrow]\!]$ to F.
3. F contains the sub-term $\lceil (\forall\ M^{\tau\to o}) \rceil$, then for every $N \in \mathcal{T}[\epsilon;\tau]$ of depth less than d add $[\![(4)\downarrow]\!]$ to F.
4. $d \leftarrow d + 1$.

Remark 2. We can in principle retain even more of the structure of λ-terms when interpreting them in the context of Z3. The support for the theory of arrays [6] in Z3 includes native handling of combinators $K : \tau \to (\tau' \Rightarrow \tau)$ (the constant array), and $map : (\tau \Rightarrow \tau') \to (\tau'' \Rightarrow \tau) \to (\tau'' \Rightarrow \tau')$ (a map combinator), besides the function $\mathsf{store} : (\tau \Rightarrow \tau') \to \tau \to \tau' \to (\tau \Rightarrow \tau')$ that updates an array at a given index. The ground theory with these combinators is decidable (satisfiability is NP complete). We call the theory CAL for combinatory array logic. We could therefore in principle extend $[\![\_]\!]$ with the cases $[\![\lambda x\ .\ M]\!] = (K\ [\![M]\!])$ if $x \notin FV(M)$, and $[\![\lambda x\ .\ (M\ (N\ x))]\!] = map([\![M]\!], [\![N]\!])$ when $x \notin FV(M) \cup FV(N)$. It would be interesting to explore to which extent CAL can be leveraged for solving HOL formulas. We could for instance prove $f \circ g = g \circ f \Rightarrow f \circ g \circ g = g \circ g \circ f$ using the decision procedure for CAL.

We implemented a light-weight HOL theorem prover based on the presented method using Z3. It is not tuned, but can (given some patience) for instance prove that injective functions have inverses: $(\forall x, y : i\ .\ (fx) = (fy)) \Longrightarrow \exists g : i \to i\ .\ \forall x : i\ .\ (g\ (f\ x)) = x$ by synthesizing the instantiation $g := \lambda x : i\ .\ (\epsilon\ (\lambda y : i\ .\ (f\ y) = x))$.

7 Conclusions

We examined a number of theories. The theories were not native to Z3, but could be either encoded using existing theories, be supported by saturating with theory axioms, or be supported efficiently using custom solvers that work in tandem with core solver. Other constraint satisfiability problems can be encoded as custom theory solvers. This includes both thoroughly and partially explored applications, such as custom constraint propagators for scheduling domains, theories with transitive closure and fixed-point operators, local theory extensions, separation logic and answer set programming.

Thanks to Chris Brown, Albert Oliveras, Nikolai Tillmann, Andrei Voronkov and Matt Dwyer for their inspiration and input on the theories and examples used here.

References

1. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Anindya Banerjee and David Naumann and Stan Rosenberg. Decision Procedures for Region Logic. *In submission*, Aug. 2011. http://www.cs.stevens.edu/ naumann/publications/dprlSubm.pdf.
3. N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In Giesl and Hähnle [9], pages 316–330.
4. C. E. Brown. Reducing higher-order theorem proving to a sequence of sat problems. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2011.

5. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The opensmt solver. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer, 2010.
6. L. de Moura and N. Bjørner. Efficient, Generalized Array Decision Procedures. In *FMCAD*. IEEE, 2009.
7. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
8. L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.
9. J. Giesl and R. Hähnle, editors. *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*. Springer, 2010.
10. J. Gil and Y. Zibin. Efficient dynamic dispatching with type slicing. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.
11. P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.
12. K. Hoder, L. Kovács, and A. Voronkov. Interpolation and symbol elimination in vampire. In Giesl and Hähnle [9], pages 188–195.
13. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
14. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in uclid. In M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2002.
15. R. Nieuwenhuis and A. Oliveras. On sat modulo theories and optimization problems. In A. Biere and C. P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2006.
16. P. Rümmer and C. Wintersteiger. Floating-point support for the Z3 SMT Solver. http://www.cprover.org/SMT-LIB-Float.
17. E. Sherman, B. J. Garvin, and M. B. Dwyer. A slice-based decision procedure for type-based partial orders. In Giesl and Hähnle [9], pages 156–170.
18. P. Suter, R. Steiger, and V. Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2011.