Satisfiability Modulo Bounded Checking

Simon Cruanes

University of Lorraine, CNRS, Inria, LORIA, 54000 Nancy, France

Abstract. We describe a new approach to find models for a computational higher-order logic with datatypes. The goal is to find counter-examples for conjectures stated in proof assistants. The technique builds on narrowing [15] but relies on a tight integration with a SAT solver to analyze conflicts precisely, eliminate sets of choices that lead to failures, and sometimes prove unsatisfiability. The architecture is reminiscent of that of an SMT solver. We present the rules of the calculus, an implementation, and some promising experimental results.

1 Introduction

Computational higher-order logics are widely used to reason about purely functional programs and form the basis of proof assistants such as ACL2 [13], Coq [9], and Isabelle [16]. Searching for models in such logics is useful both for refuting wrong conjectures and for testing — it is often faster and easier to test a property than to prove it. In this work we focus on a logic with algebraic datatypes and terminating recursive functions. Once proven terminating, these functions have a natural interpretation in any model as least fixpoints.

The typical use case is for the users to specify a property they believe to hold for the program they wrote and let a solver search for a (counter-)example until some resource is exhausted — time, patience, etc. Our goal is to build a tool that can be used for finding counter-examples in proof assistants. Figure 1 presents such a problem in TIP syntax [6] that defines natural numbers, lists, and operations on lists, where the (unsatisfiable) goal is to find a list of natural numbers that is a palindrome of length 2 with sum 3.

In the functional programming community, tools such as QuickCheck [5] and SmallCheck [19] have been used to test conjectures against random values or up to a certain depth. Feat [11] is similar to SmallCheck but enumerates inputs by increasing size, rather than depth. However, QuickCheck is limited when invariants have to be enforced (e.g. red-blackness of trees), forcing users to write custom random generators, and SmallCheck and Feat can get lost quickly in large search spaces. Lazy SmallCheck (LSC) is similar to SmallCheck but relies on the lazy semantics of Haskell to avoid enumerating inputs that are not needed to evaluate the property. LSC is close to narrowing [1, 15], a symbolic approach that has ties to functional logic programming [12] and builds a model incrementally. Nevertheless, LSC and narrowing-based tools explore the space of possible inputs quite naively, making many counter-examples very hard to find. All these approaches lack a way of analyzing why a given search path failed.

```
(declare-datatypes () ((Nat (Z) (S (prec Nat)))))
(declare-datatypes () ((List (Nil) (Cons (hd Nat) (tl List)))))
(define-fun-rec plus ((x Nat) (y Nat)) Nat
  (match x (case Z y) (case (S x2) (S (plus x2 y)))))
; some definitions omitted
(define-fun-rec rev ((l List)) List
  (match l (case Nil Nil) (case (Cons x l2) (append (rev l2) (Cons x Nil)))))
(assert-not (forall ((l List))
  (not (and (= l (rev l)) (= (length l) (S (S Z))) (= (sum l) (S (S (S Z))))))))
```

Fig. 1. Looking for impossible palindromes

Modern SMT solvers are often efficient in difficult combinatorial problems. They rely on a SAT solver to analyze conflicts and interleave theory reasoning with propositional choices. However, their focus is first-order classical logic, where symbols are neatly partitioned between theory symbols that have a precise definition and user-provided symbols that are axiomatized. When a user want to introduce their own parameterized operators, they must use quantifiers and full first order logic, where solvers are usually incomplete. Some work has been done on handling datatypes [4, 17] and recursive functions in SMT solvers such as CVC4 [18] or calling an SMT solver repeatedly while expanding function definitions as in Leon [20], but each reduction step (e.g. function call) is very expensive. Many tools dedicated to analysing imperative programs, where control flow is based on if/then/else, check the satisfiability of a given bad path using a SMT solver, but as in Leon the expansion of loops and functions is done from outside the SMT solver itself.

Bridging the gap between QuickCheck and SMT solvers is HBMC [8] (Haskell Bounded Model Checker — not published yet). HBMC progressively encodes the evaluation graph into propositional constraints (effectively "bit-blasting" recursive functions and datatypes), leveraging the powerful constraint propagations of modern SAT solvers. However, it suffers from the same weakness as SMT-based techniques: every evaluation step has to be encoded, then performed, inside the SAT solver, making computations slow.

We present a new technique, *Satisfiability Modulo Bounded Checking* (SMBC) that occupies a middle ground between narrowing and HBMC. On the one hand, it can evaluate terms more efficiently than pure bit-blasting although not quite as fast as native code; on the other hand it benefits from propositional conflict-driven clause learning (CDCL) of modern SAT solvers to never make the same bad choice twice. Two main components are involved: (i) a symbolic evaluation engine (Sect. 3), and (ii) a SAT solver with incremental solving under assumptions (Sect. 4). Those two components communicate following lazy SMT techniques [3]. Inputs are lazily and symbolically enumerated using *iterative deepening* (Sect. 5) to ensure fairness, but we use the ability of the SAT solver to solve under assumptions to avoid the costly re-computations usually associated with that technique. In addition, building on CDCL allows SMBC to sometimes

prove the unsatisfiability of the problem, something evaluation-based tools are incapable of.

We can extend SMBC to support uninterpreted types and unspecified functions (Sect. 6). After presenting refinements to the calculus (Sect. 7) and an implementation (Sect. 8), we run some experiments (Sect. 9) to compare SMBC with some of the previously mentioned tools on various families of problems. Detailed proofs and additional content can be found in our report.[1]

2 Logic

We consider a multi-sorted higher-order classical logic, without polymorphism. A finite set of mutually recursive *datatypes* $d_1, \ldots, d_k$ is defined by a system

$$\left(d_i \stackrel{\text{def}}{=} c_{i,1}(\overline{\alpha_{i,1}}) \mid \cdots \mid c_{i,n_i}(\overline{\alpha_{i,n_i}})\right)_{i \in \{1,\ldots,k\}}$$

where the $\overline{\alpha_{i,j}}$ are tuples of type arguments. We consider only *standard models*, in which the domain of a datatype is the set of terms freely generated from its constructors. Similarly, mutually recursive *functions* $f_1, \ldots, f_k$ are defined by a set of equations $f_1(\overline{x_1}) \stackrel{\text{def}}{=} t_1, \ldots, f_k(\overline{x_k}) \stackrel{\text{def}}{=} t_k$ that we assume total and terminating. The term language comprises bound variables, datatype constructors, shallow pattern-matching over datatypes, λ-abstractions $\lambda x : \tau.\ t$, and applications $f\ t_1\ \ldots\ t_n$. Constructors are always fully applied. $t\sigma$ is the application of a substitution over bound variables σ to t. $\mathsf{Bool} \stackrel{\text{def}}{=} \{\top, \bot\}$ is a special datatype, paired with tests $\mathsf{if}\ a\ b\ c$ that are short for $\mathsf{case}\ a\ \mathsf{of}\ \top \to b \mid \bot \to c\ \mathsf{end}$. A *value* is a λ-abstraction or constructor application. The operators $\wedge : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ and $\neg : \mathsf{Bool} \to \mathsf{Bool}$ have the usual logical semantics; evaluation of $\wedge$ is parallel rather than the usual sequential semantics it has in most programming languages: $t \wedge \bot$ reduces to $\bot$ even if t is not a value. We will speak of *parallel conjunction*. Other boolean connectives are encoded in terms of $\wedge$ and $\neg$. We also define an ad hoc polymorphic equality operator $=$ that has the classic structural semantics on datatypes and booleans; comparison of functions is forbidden. An *unknown* is simply an uninterpreted constant which must be given a value in the model. This logic corresponds to the monomorphic fragment of TIP [6] or the extension of SMT-LIB [2] with recursive functions, with the additional assumption that they always terminate.

A *data value* is a term built only from constructor applications, bound variables, and λ-abstractions (without defined symbols, matching, or unknowns). The *depth* of a data value is recursively defined as 1 on constant constructors, $1 + \mathsf{depth}(t)$ for $\lambda x.\ t$, and $1 + \max_{i=1\ldots n} \mathsf{depth}(t_i)$ on constructor applications $c(t_1, \ldots, t_n)$.[2] A *goal set* G is a set of boolean terms. A *model* of G is a mapping

[1] `https://cedeela.fr/~simon/files/cade_17_report.pdf`

[2] A more flexible definition $\mathsf{depth}(c(t_1, \ldots, t_n)) = \mathsf{cost}(c) + \max_{i=1\ldots n} \mathsf{depth}(t_i)$ can also be used to skew the search towards some constructors, as long as $\mathrm{cost}(c) > 0$ holds for all c.

from unknowns of G to data values, such that $\bigwedge_{t \in G} t$ evaluates to $\top$. The depth of a model is the maximal depth of the data values in it.

In the rest of this paper, t, u will represent terms, k will be unknowns, c, d will be constructors, and e will stand for explanations (conjunctions of literals). We will use an injective mapping to propositional variables denoted $[\![\cdot]\!]$.

3 Evaluation with Explanations

The semantics of our logic relies on evaluating expressions that contain tests, pattern matching, and (recursive) functions. Because expressions can contain unknowns, their reduction is influenced by assignments to these unknowns. We need an evaluator that keeps track of which choices were used to reduce a term. In this way, when a goal term reduces to $\bot$, we know that this combination of choices is wrong.

In Figure 2, we show the evaluation rules for terms, given a substitution ρ on unknowns. The notation $t \xrightarrow{\rho}_e u$ means that t reduces to u in one step, with explanations e (a set of boolean literals), under substitution ρ. We denote $t \xrightarrow{\rho}{}^*_e u$ for the transitive reflexive closure of the reduction. We write $t\downarrow_\rho$ (the *normal form* of t under ρ) for the unique term u such that $t \xrightarrow{\rho}{}^*_e u$ and no rule applies to u. In a first approximation, ignoring the explanations, the rules correspond to a normal call-by-need evaluation strategy for the typed λ-calculus. This matches the definition of values given earlier: a value is a weak head normal form. It is possible to use environments instead of substitutions, carrying bindings in every rule, but we chose this presentation for reasons related to hash-consing, as often used in SMT solvers. The choice of call-by-need rather than call-by-value is justified by the maximal amount of laziness it provides in presence of unknowns: instead of waiting for function call arguments, matched terms, or test conditions to be fully evaluated (and therefore, for their unknowns to be fully decided in the partial model), we can proceed with only a weak head normal form.

The rules **id** and **trans** specify how explanations are combined in the reflexive transitive closure; The rule **case** reduces a pattern matching once the matched term is a value (i.e. starts with a constructor, by typing). The rule **app** allows to reduce the function term in an application (until it becomes a value, that is, a λ-abstraction); rule β is the regular β-reduction; rule **def** unfolds definitions (in particular, recursive definitions are unfolded on demand). The rule **decision** replaces an unknown with its value in the current substitution ρ (i.e. the partial model). The other rules define the semantics of boolean operators and equality. We forbid checking equality of functions as is it not computable.

Whether to use small-step or big-step semantics (i.e. reducing a term by one step if a subterm reduces, or waiting for the subterm to become a value) is of little importance for most cases. The only exception is the rules for conjunction, in which big-step semantics is required (i.e. $a \wedge b$ does not always reduce when, e.g., a reduces). To see why, assume small-step semantics and consider $a \xrightarrow{\rho}{}^*_{e_1} a' \xrightarrow{\rho}{}^*_{e_3} \bot$ and $b \xrightarrow{\rho}{}^*_{e_2} b'$ where $a, b : \mathsf{Bool}$. The following reduction

$$a \wedge b \xrightarrow{\rho}{}^*_{e_1} a' \wedge b \xrightarrow{\rho}{}^*_{e_1 \cup e_2} a' \wedge b' \xrightarrow{\rho}{}^*_{e_1 \cup e_2 \cup e_3} \bot \wedge b' \xrightarrow{\rho}{}^*_{e_1 \cup e_2 \cup e_3} \bot$$

$$\frac{}{a \xrightarrow{\rho}_{\emptyset}^{*} a}\ \textbf{id} \qquad \frac{a \xrightarrow{\rho}_{e_1} b \qquad b \xrightarrow{\rho}_{e_2}^{*} c}{a \xrightarrow{\rho}_{e_1 \cup e_2}^{*} c}\ \textbf{trans}$$

$$\frac{c \text{ is a constructor} \qquad t \xrightarrow{\rho}_{e}^{*} c(t_1, \ldots, t_n)}{\mathsf{case}\ t\ \mathsf{of}\ c(x_1, \ldots, x_n) \to u \mid \cdots\ \mathsf{end} \xrightarrow{\rho}_{e} u[t_1/x_1, \ldots, t_n/x_n]}\ \textbf{case}$$

$$\frac{f \xrightarrow{\rho}_{e} g}{f\ t \xrightarrow{\rho}_{e} g\ t}\ \textbf{app} \qquad \frac{}{(\lambda x.\ t)\ u \xrightarrow{\rho}_{\emptyset} t[u/x]}\ \beta \qquad \frac{x \stackrel{\text{def}}{=} t}{x \xrightarrow{\rho}_{\emptyset} t}\ \textbf{def}$$

$$\frac{\rho(k) = t}{k \xrightarrow{\rho}_{\{\![k:=t]\!\}} t}\ \textbf{decision} \qquad \frac{a \xrightarrow{\rho}_{e}^{*} \bot}{a \wedge b \xrightarrow{\rho}_{e} \bot}\ \textbf{and-left}$$

$$\frac{b \xrightarrow{\rho}_{e}^{*} \bot}{a \wedge b \xrightarrow{\rho}_{e} \bot}\ \textbf{and-right} \qquad \frac{a \xrightarrow{\rho}_{e_a}^{*} \top \qquad b \xrightarrow{\rho}_{e_b}^{*} \top}{a \wedge b \xrightarrow{\rho}_{e_a \cup e_b} \top}\ \textbf{and-true}$$

$$\frac{a \xrightarrow{\rho}_{e}^{*} \top}{\neg a \xrightarrow{\rho}_{e} \bot}\ \textbf{not-true} \qquad \frac{a \xrightarrow{\rho}_{e}^{*} \bot}{\neg a \xrightarrow{\rho}_{e} \top}\ \textbf{not-false}$$

$$\frac{a \xrightarrow{\rho}_{e} a'}{a = b \xrightarrow{\rho}_{e} a' = b}\ \textbf{eq-left} \qquad \frac{b \xrightarrow{\rho}_{e} b'}{a = b \xrightarrow{\rho}_{e} a = b'}\ \textbf{eq-right}$$

$$\frac{c, d \text{ are constructors} \qquad c \neq d}{c(\overline{t}) = d(\overline{u}) \xrightarrow{\rho}_{\emptyset} \bot}\ \textbf{eq-conflict}$$

$$\frac{c \text{ is a constructor}}{c(t_1, \ldots, t_n) = c(u_1, \ldots, u_n) \xrightarrow{\rho}_{\emptyset} \bigwedge_{i=1}^{n} t_i = u_i}\ \textbf{eq-sub}$$

Fig. 2. Evaluation rules under substitution ρ

is imprecise because e_2 is not actually needed for $a \wedge b \xrightarrow{\rho}^{*} \bot$, $e_1 \cup e_3$ is sufficient. The resulting explanation is not as general as it could be, and a smaller part of the search space will be pruned as a result.

Evaluation of a normal form t that is not a value in a substitution ρ is *blocked* by a set of unknowns $\mathsf{block}_\rho(t)$:

$$\begin{aligned}
\mathsf{block}_\rho(\lambda x.\ t) &= \emptyset \\
\mathsf{block}_\rho(c(u_1, \ldots, u_n)) &= \emptyset \text{ if } c \text{ is a constructor} \\
\mathsf{block}_\rho(f\ t) &= \mathsf{block}_\rho(f) \\
\mathsf{block}_\rho(\mathsf{case}\ t\ \mathsf{of}\ \ldots\ \mathsf{end}) &= \mathsf{block}_\rho(t{\downarrow_\rho}) \\
\mathsf{block}_\rho(k) &= \{k\} \text{ if } k \text{ is an unknown} \\
\mathsf{block}_\rho(a = b) &= \mathsf{block}_\rho(a) \cup \mathsf{block}_\rho(b) \\
\mathsf{block}_\rho(\neg a) &= \mathsf{block}_\rho(a{\downarrow_\rho}) \\
\mathsf{block}_\rho(a \wedge b) &= \mathsf{block}_\rho(a{\downarrow_\rho}) \cup \mathsf{block}_\rho(b{\downarrow_\rho})
\end{aligned}$$

In some cases, the blocking unknowns are found in the normal form of subterms of t. This corresponds to the evaluation rules that wait for the subterm to become a value before reducing.

Lemma 1 (Uniqueness of values for $\xrightarrow{\rho}{}^*$). *If $t \xrightarrow{\rho}{}^*_{e_1} v_1$ and $t \xrightarrow{\rho}{}^*_{e_2} v_2$ where v_1 and v_2 are values, then $v_1 = v_2$.*

Proof. The rules are deterministic, and values are always normal forms since no rule applies to them. □

Lemma 2. *If $t = t{\downarrow_\rho}$ is a normal form, then $\mathsf{block}_\rho(t) = \emptyset$ iff t is a value.*

Proof. By induction on the shape of t. If t is a value, then it is a normal form as no rule rewrites a normal form. Conversely, if $\mathsf{block}_\rho(t) = \emptyset$, either t is a value, or t is an application, test, negation, equality or conjunction whose reductive subterms are themselves blocked by $\emptyset$, by definition of $\mathsf{block}_\rho(\cdot)$.

- In the *app* case, if f is a value then β-reduction would apply and t would reduce, absurd. Therefore f is not a value. If f is in normal form but not a value, by inductive hypothesis its blocking set is not empty, which is absurd. Then f is not in normal form, but then $f \xrightarrow{\rho}{}^*_e f'$ and t would reduce too, absurd. Therefore t must be a value.
- In the *eq* case, either a and b are values, meaning t should reduce; or (wlog) a is not a value. Then, same as the *app* case, a being in normal form contradicts $\mathsf{block}_\rho(a) = \emptyset$, and a not being in normal form contradicts the irreducibility of t.
- In the *case* and *not* cases, consider the normal form u of the immediate subterm. This normal form is not a value (t would reduce); therefore by inductive hypothesis $\mathsf{block}_\rho(u) \neq \emptyset$ implying $\mathsf{block}_\rho(t) \neq \emptyset$, absurd.
- In the *and* case, the argument is similar, but with two subterms instead of one. At least one of these subterms does not reduce to a value.

□

4 Delegating Choices and Conflict Analysis to SAT

We now have evaluation rules for reducing terms given a substitution on unknowns but have not yet explained how this substitution is built. As in narrowing [1, 15], it is constructed by *refining* unknowns incrementally, choosing their head constructor (or boolean value) and applying it to new unknowns that might need to be refined in turn if they block evaluation.[3] However, in our case, the SAT solver will do the refinement of an unknown k once it has been *expanded*; the first time $k : \tau$ blocks the evaluation of a goal g (i.e., $k \in \mathsf{block}_\rho(g)$), some clauses are added to the SAT solver, forcing it to satisfy exactly one of the literals $[\![k := c_i(k_{i,1}, \ldots, k_{i,n_i})]\!]$, where c_i is a constructor of τ. Once one of the literals $[\![k := t_i]\!]$ is true in the SAT solver's partial model — implying that $\rho(k) = t_i$, as we will see next — evaluation of the goal g can resume using rule **decision** (in Figure 2) and k is no longer blocking.

The state of the SAT solver is represented below as a pair $M \parallel F$ where M is the trail (a set of literals not containing both l and $\neg l$), and F is a set of clauses. The operation $\mathsf{subst}(M)$ extracts a substitution on unknowns from positive literals in the trail:

$$\mathsf{subst}(M)(k) = t \quad \text{if} \quad [\![k := t]\!] \in M$$

It is a valid substitution because for each k there is at most one such literal that is true at any time.

The interactions between the SAT solver and the evaluation engine are bidirectional. When the SAT solver makes some decisions and propagations, yielding the new state $M \parallel F$, the substitution $\mathsf{subst}(M)$ is used to evaluate the goals in G. If all the goals evaluate to $\top$, we can report M as a model. Otherwise, if there is a goal $t \in G$ such that $t \xrightarrow{\mathsf{subst}(M)}{}^{*}_{e} \bot$, M must be discarded. This is done by adding to F a *conflict clause* $C \stackrel{\text{def}}{=} \bigvee_{a \in e} \neg a$ that blocks the set of choices in e. The SAT solver will *backjump* to explore models not containing e. Backjumping with clause C and state $M \parallel F$ returns to a state $M' \parallel F$ where M' is the longest prefix of M in which C is not absurd. We do not detail the precise mechanism of *clause learning* that is used for conflict resolution — it is identical to regular T-Backjump in Barrett et al. [3].

More formally, Figure 3 recalls some of the rules driving the SAT solver and the evaluation engine from Section 3, following the style of Barrett et al. We add the new rule **goal conflict** (below), which instantiates T-Backjump in Barrett et al. G is the global set of goals (boolean terms) to satisfy. A literal l^d designates a *decision* literal, that is, a choice point (by opposition to *propagations*).

This mechanism allows to call the evaluation function with a new substitution after every boolean decision (and propagation), so that flawed partial models are discovered as soon as possible. It is also possible to wait until the SAT solver has constructed a complete boolean model before reducing goals, making both components more loosely coupled, but at the price of late detection of failures.

[3] Our framework corresponds to the special case of needed narrowing when the only rewrite rules are those defining pattern matching.

$$\frac{M \parallel F \qquad \bigwedge_{g \in G} g \xrightarrow{\mathsf{subst}(M)}{}^{*}_{e} \bot}{M \parallel F \leftarrow \text{backjump using conflict clause } \bigvee_{a \in e} \neg a} \textbf{ goal conflict}$$

$$\frac{M \parallel F, C \vee l \qquad M \models \neg C \qquad l \text{ undefined in } M}{M, l \parallel F, C \vee l} \textbf{ unit propagate}$$

$$\frac{M \parallel F \qquad l \text{ occurs in } F \qquad l \text{ undefined in } M}{M, l^d \parallel F} \textbf{ decide}$$

$$\frac{M \parallel F, C \qquad M \models \neg C}{\text{backjump from } C} \textbf{ boolean conflict}$$

$$\frac{M \parallel F, C \qquad M \models \neg C \qquad C \text{ contains no decision literal}}{\text{return } \textsc{Unsat}} \textbf{ unsat}$$

Fig. 3. Rules for the SAT solver

Lemma 3 (Monotonicity of Models). *A model of G, expressed as a trail M, satisfies $\bigwedge_{t \in G} t \xrightarrow{\mathit{subst}(M)}{}^{*}_{e} \top$. No subset of M reduces $\bigwedge_{t \in G} t$ to $\bot$.*

Proof. A model M, by definition, reduces the goals to $\top$. Now, considering one of its subsets $N \subseteq M$, we prove that $\bigwedge_{t \in G} t \xrightarrow{\mathsf{subst}(N)}{}^{*}_{e} \bot$ is impossible, *ab absurdo*. For any $t \in G$, either $t{\downarrow_{\mathsf{subst}(N)}} = t{\downarrow_{\mathsf{subst}(M)}} = \top$ (absurd), or $t{\downarrow_{\mathsf{subst}(N)}}$ is not fully reduced because it is blocked by a non-empty subset of the unknowns bound in $\mathsf{subst}(M)$ but not in $\mathsf{subst}(N)$. It cannot be $\bot$, because

$$\bot \xleftarrow{* \ \mathsf{subst}(N)} t \xrightarrow{\mathsf{subst}(N)}{}^{*} t{\downarrow_{\mathsf{subst}(N)}} \xrightarrow{\mathsf{subst}(M)}{}^{*}_{e'} \top$$

contradicts the uniqueness property of $\longrightarrow^*$ (Lemma 1).

5 Enumeration of Inputs and Iterative Deepening

We have not specified precisely how to enumerate possible models. This section presents a fair enumeration strategy based on Iterative Deepening [14].

A major issue with a straightforward combination of our evaluation function and SAT solver is that there is a risk of non-termination. Indeed, a wrong branch might never be totally closed. Consider the goal $p(b) \wedge a + b = \mathrm{Z}$ with unknowns $\{a, b\}$, where $p(x) \stackrel{\text{def}}{=} \mathsf{case}\ x\ \mathsf{of}\ \mathrm{Z} \to \top \mid \mathrm{S}(\_) \to \top\ \mathsf{end}$ is trivial, and $+$ is defined on Peano numbers by recursion on its left argument. Then making the initial choice $b = \mathrm{S}(b_2)$ (to unblock $p(b)$) and proceeding to refine a in order to unblock $a + b = \mathrm{Z}$ will lead to an infinite number of failures related to a, none of which will backjump past $b = \mathrm{S}(b_2)$.

To overcome this issue, we solve a series of problems where the *depth* of unknowns is limited to increasingly large values, a process inspired from iterative

deepening. Because the SAT solver controls the shape of unknowns, we use special boolean literals $[\![\mathsf{depth} \leq n]\!]$ to forbid any choice that causes an unknown to be deeper than n; then we solve under assumption $[\![\mathsf{depth} \leq n]\!]$. If a model is found, it is also valid without the assumption and can be returned immediately to the user. Otherwise, we need the SAT solver to be able to provide *unsat cores* — the subset of its clauses responsible for the problem being unsatisfiable — to make the following distinction: if $[\![\mathsf{depth} \leq n]\!]$ contributed to the unsat core, it means that there is no solution within the depth limit, and we start again with $[\![\mathsf{depth} \leq n + \textsc{Step}]\!]$ (where $\textsc{Step} \geq 1$). The last case occurs when the conflict does not involve the assumption $[\![\mathsf{depth} \leq n]\!]$: then the problem is truly unsatisfiable (e.g., in Figure 1).

The iterative deepening algorithm is detailed below, in three parts: (i) the main loop, in Algorithm 1; (ii) solving within a depth limit, in Algorithm 2; (iii) expanding unknowns, in Algorithm 3. These functions assume that the SAT solver provides functions for adding clauses dynamically (AddSatClause), adding a conflict clause (Conflict), performing one round of decision then boolean propagation (MakeSatDecision and BoolPropagate), and extracting unsat cores (UnsatCore). These functions modify the SAT solver state $M \parallel F$. In practice, it is also possible to avoid computing unsat cores at line 7 in Algorithm 1, by checking for pure boolean satisfiability again, but without the depth-limit assumption. Most computations (including the current normal form of $\bigwedge_{t \in G} t$) can be done incrementally and are backtracked in case of conflict.

Algorithm 1 Main Loop Using Iterative Deepening

Require: $\textsc{Step} \geq 1$: depth increment, G: set of goals

```
1: function MainLoop(G)
2:     d ← Step                                          ▷ initial depth
3:     while d ≤ MaxDepth do
4:         res ← SolveUpTo(G, d)
5:         if res = Sat then return Sat
6:         else if [[depth ≤ d]] ∉ UnsatCore(res) then return Unsat
7:         else d ← d + Step
8:     return Unknown
```

Theorem 1 (Termination). *The function* SolveUpTo *in Algorithm 2 terminates.*

Proof. There are only a finite number of unknowns that will be expanded for a given d (corresponding, at most, to the full space of input values of depth $\leq d$) because the other choices are forbidden by expansion. There also are a finite number of possible conflicts for a given d (at most the number of input values of depth $\leq d$), since clause learning prevents the same bad choice to be made twice. At each loop iteration, either a new conflict is found, or a decision on one unknown is made, progressing towards a partial model. The pair (n_c, n_d), where

Algorithm 2 Solving Within a Depth Limit

Require: G: set of goal terms, d: depth limit

1: **function** SolveUpTo(G, d)
2: AddAssumption($[\![\mathsf{depth} \leq d]\!]$) ▷ local assumption
3: $M \parallel F \leftarrow \emptyset \parallel G$ ▷ initial model and clauses
4: **while** true **do**
5: $M \parallel F \leftarrow$ MakeSatDecision($M \parallel F$) ▷ model still partial
6: $M \parallel F \leftarrow$ BoolPropagate($M \parallel F$)
7: $G' \leftarrow \{(u, e) \mid t \in G, t \xrightarrow{\mathsf{subst}(M)}{}^{*}_{e} u\}$ ▷ current normal form of G
8: **if** $(\bot, e) \in G'$ **then**
9: $M \parallel F \leftarrow$ Conflict($M \parallel F \cup \{\bigvee_{a \in e} \neg a\}$) ▷ backjump or Unsat
10: **else if** all terms in G' are $\top$ **then return** Sat
11: **else**
12: $B \leftarrow \bigcup_{(t,e) \in G'} \mathsf{block}_{\mathsf{subst}(M)}(t)$ ▷ blocking unknowns
13: **for** $k \in B$, k not expanded **do**
14: $F \leftarrow F \cup$ Expand(k, d) ▷ will add new literals and clauses

Algorithm 3 Expansion of Unknowns

Require: k: unknown of type τ, d: depth limit

1: **function** Expand(k, d)
2: **let** $\tau = c_1(\tau_{1,1}, \ldots, \tau_{1,n_1}) \mid \ldots \mid c_k(\tau_{k,1}, \ldots, \tau_{k,n_k})$
3: $l \leftarrow \{c_i(k_{i,1}, \ldots, k_{i,n_i}) \mid i \in 1, \ldots, k\}$ ▷ each $k_{i,j:\tau_{i,j}}$ is a fresh unknown
4: AddSatClause($\bigvee_{t \in l} [\![k := t]\!]$)
5: AddSatClauses($\{\neg[\![k := t_1]\!] \vee \neg[\![k := t_2]\!] \mid (t_1, t_2) \in l, t_1 \neq t_2\}$)
6: **for** $t \in l$ **where** depth$(t) > d$ **do**
7: AddSatClause($\neg[\![\mathsf{depth} \leq d]\!] \vee \neg[\![k := t]\!]$) ▷ block this choice at depth d

n_c is the number of conflicts yet to find and n_d the number of decisions to make in current branch, therefore decreases at each loop iteration. □

Theorem 2 (Soundness). *The function* SolveUpTo *in Algorithm 2 returns either* Sat *or* Unsat*. If it returns* Sat*, then the substitution* $\mathsf{subst}(M)$ *from the boolean trail is a model. If it returns Unsat, then there are no solutions of depth smaller than* d*.*

Proof. In case of Sat, at line 10, all goals evaluate to $\top$ under $\mathsf{subst}(M)$, therefore it is a model. Conversely, assume there is a model of G of depth smaller than d; this model must satisfy the following sets of clauses: (i) S_k (clauses from expansion of unknowns, all are tautologies of a conservative extension of the theory of datatypes); (ii) $S_\bot$ (conflict clauses); (iii) and $S_a = \{[\![\mathsf{depth} \leq d]\!]\}$ which is the local assumption. This last clause is true only in models that are within the depth limit. There must be a boolean model of those clauses, which means that the SAT solver will at some point find such a boolean model M; by definition, $\bigwedge_{t \in G} t \xrightarrow{\mathsf{subst}(M)}{}^{*} \top$. By Lemma 3, no prefix of M reduces G to $\bot$, meaning that the evaluation engine does not forbid the model by adding a conflict clause;

eventually, M must be returned, and the solver returns SAT. Therefore, returning UNSAT only happens if there is no sufficiently shallow model. □

Theorem 3 (Bounded Completeness). *If there exists a model of depth smaller at most* STEP $\lfloor$MAXDEPTH/STEP$\rfloor$*, then Algorithm 1 will return* SAT.

Proof. The depth d is always a multiple of STEP. Let $d_{\min} \leq$ MAXDEPTH be the smallest multiple of STEP such that there is a model of depth $\leq d_{\min}$. Iterations of the loop with $d < d_{\min}$ return UNSAT by soundness of SOLVEUPTO (Theorem 2); the iteration at depth $d_{\min}$ returns SAT. □

5.1 Application to the Introductory Example

We illustrate our technique on an example.[4] Pick the same definitions as in Figure 1, but with the goal set $G \stackrel{\text{def}}{=} \{\mathsf{rev}(l) = l,\ \mathsf{length}(l) = 2,\ \mathsf{sum}(l) = 2\}$ where the unknown is a list l. Unlike in Figure 1, this problem is satisfiable. Assuming STEP $= 1$, we start solving under constraint $[\![\mathsf{depth} \leq 1]\!]$. Under the empty substitution, G reduces to a set of terms all blocked by a pattern matching on l; expansion of l into $\{\mathrm{Nil},\ \mathrm{Cons}(x_1, l_1)\}$ follows, where $x_1 : \mathsf{Nat}$ and $l_1 : \mathsf{List}$ are fresh unknowns. Suppose the SAT solver picks $[\![l := \mathrm{Nil}]\!]$. G reduces to $\{\top, \bot, \bot\}$ with explanations $\{[\![l := \mathrm{Nil}]\!]\}$, so the conflict clause $\neg[\![l := \mathrm{Nil}]\!]$ is asserted, added to the partial model with no effect, and the solver backtracks.

The next boolean decision must be $[\![l := \mathrm{Cons}(x_1, l_1)]\!]$. Subsequently, G reduces to

$$\{\mathsf{append}(\mathsf{rev}(l_1), \mathrm{Cons}(x_1, \mathrm{Nil})) = \mathrm{Cons}(x_1, l_1),\ \mathsf{length}(l_1) = 1,\ x_1 + \mathsf{sum}(l_1) = 2\}$$

(more precisely, to a less readable version of these terms where function definitions are unfolded into some pattern matching). The resulting set is blocked both by l_1 (in the first two terms) and x_1 (in $x_1 + \mathsf{sum}(l_1)$). Expansion of these terms yields $\{\mathrm{Nil},\ \mathrm{Cons}(x_2, l_2)\}$ and $\{\mathrm{Z},\ \mathrm{S}(y_1)\}$, but the choice $\mathrm{Cons}(x_2, l_2)$ is blocked by $[\![\mathsf{depth} \leq 1]\!]$. The solver must choose $[\![l_1 := \mathrm{Nil}]\!]$, which entails

$$\mathsf{length}(l) = 2 \xrightarrow{\rho}{}^{*}_{\{[\![l:=\mathrm{Cons}(x_1,l_1)]\!]\}} \mathsf{length}(l_1) = 1 \xrightarrow{\rho}{}^{*}_{\{[\![l_1:=\mathrm{Nil}]\!]\}} 0 = 1 \xrightarrow{\rho}{}^{*}_{\emptyset} \bot$$

The conflict clause $\neg[\![l := \mathrm{Cons}(x_1, l_1)]\!] \vee \neg[\![l_1 := \mathrm{Nil}]\!]$ triggers an UNSAT result, but only because of the assumption $[\![\mathsf{depth} \leq 1]\!]$.

The main loop (Algorithm 1) then proceeds to depth 2, and tries to solve the problem again under the assumption $[\![\mathsf{depth} \leq 2]\!]$. The SAT solver can now pick $[\![l_1 := \mathrm{Cons}(x_2, l_2)]\!]$. At some point, it will pick $[\![l_2 := \mathrm{Nil}]\!]$ (the other choice is too deep), $[\![x_1 := \mathrm{S}(y_1)]\!]$, $[\![y_1 := \mathrm{Z}]\!]$, $[\![x_2 := \mathrm{S}(y_2)]\!]$, and $[\![y_2 := \mathrm{Z}]\!]$. Other choices would reduce one of the goals to $\bot$: once the shape of l_1 is fixed by the length constraint, $\mathsf{rev}(l) = l$ reduces to $x_1 = x_2$ and $\mathsf{sum}(l) = 2$ becomes $x_1 + x_2 = 2$, and wrong choices quickly reduce those to $\bot$. At this point, $\bigwedge_{t \in G} t \xrightarrow{\rho}{}^{*} \top$ and we obtain the model $l = \mathrm{Cons}(1, \mathrm{Cons}(1, \mathrm{Nil}))$.

[4] The example is provided at `https://cedeela.fr/~simon/files/cade_17.tar.gz` along with other benchmarks.

6 Extensions of the Language

6.1 Uninterpreted Types

Finding counter-example for programs and formalizations that use only recursive function definitions might still involve uninterpreted types arising from type Skolemization or abstract types. To handle those in SMBC, e.g. for a type τ, which corresponds to a finite set of domain elements denoted $\mathsf{elt}_0(\tau), \mathsf{elt}_1(\tau), \ldots$ (domain elements behave like constructors for evaluation). We also introduce *type slices* $\tau_{[0\ldots]}, \tau_{[1\ldots]}, \ldots$ where $\tau \stackrel{\text{def}}{=} \tau_{[0\ldots]}$. Conceptually, a type slice $\tau_{[n\ldots]}$ corresponds to the subtype of τ that excludes its n first elements: $\tau_{[n\ldots]} \stackrel{\text{def}}{=} \{\mathsf{elt}_n(\tau), \ldots, \mathsf{elt}_{\mathrm{card}(\tau)-1}(\tau)\}$. Then, we introduce propositional literals $[\![\mathsf{empty}(\cdot)]\!]$ that will be given a truth value by the SAT solver; if $[\![\mathsf{empty}(\tau_{[n\ldots]})]\!]$ is true, it means $\tau_{[n\ldots]} \equiv \emptyset$; otherwise, it means $\tau_{[n\ldots]} \equiv \{\mathsf{elt}_n(\tau)\} \cup \tau_{[n+1\ldots]}$. We assume $\neg[\![\mathsf{empty}(\tau_{[0\ldots]})]\!]$. Expansion of some unknown $k : \tau_{[n\ldots]}$ yields the following boolean constraints:

$$\begin{aligned}
[\![\mathsf{empty}(\tau_{[n-1\ldots]})]\!] &\Rightarrow [\![\mathsf{empty}(\tau_{[n\ldots]})]\!] \\
[\![\mathsf{depth} \leq n]\!] &\Rightarrow [\![\mathsf{empty}(\tau_{[n\ldots]})]\!] \\
[\![k = \mathsf{elt}_n(\tau)]\!] &\vee \left(\neg[\![\mathsf{empty}(\tau_{[n+1\ldots]})]\!] \wedge [\![k := k']\!]\right)
\end{aligned}$$

where $k' : \tau_{[n+1\ldots]}$ is a fresh unknown belonging in the next slice of τ. To express constraints on τ, the input language provides finite quantifiers $\forall x : \tau.\ F$ and $\exists x : \tau.\ F$ (which abbreviates $\neg(\forall x : \tau.\ \neg F)$). The quantifier is interpreted with the following rules:

$$\frac{\rho([\![\mathsf{empty}(\tau_{[n\ldots]})]\!]) = \top}{\forall x : \tau_{[n\ldots]}.\ F \xrightarrow{\rho}_{\{[\![\mathsf{empty}(\tau_{[n\ldots]})]\!]\}} \top}\ \textbf{forall-empty}$$

$$\frac{\rho([\![\mathsf{empty}(\tau_{[n\ldots]})]\!]) = \bot}{\forall x : \tau_{[n\ldots]}.\ F \xrightarrow{\rho}_{\{\neg[\![\mathsf{empty}(\tau_{[n\ldots]})]\!]\}} F[\mathsf{elt}_n(\tau)/x] \wedge \left(\forall x : \tau_{[n+1\ldots]}.\ F\right)}\ \textbf{forall-pair}$$

remark It is possible to emulate uninterpreted types by defining them as a datatype isomorphic to Peano natural numbers (0 or successor), and then defining a special $\mathrm{card}(\tau) : \tau$ (with $\mathrm{card}(\tau) \neq 0$) that acts as a higher bound to any other value of this type. However, this requires defining the special predicate $<$ and adding a new constraint $k < \mathrm{card}(\tau)$ for every unknown $k : \tau$; the implementation would not be simpler.

6.2 Functional Unknowns

With uninterpreted types often come functions taking arguments of uninterpreted types. We can also wish to synthesize (simple) functions taking booleans or datatypes as parameters. It is possible to build functions by *refinement*, using currying (considering only one argument at a time) depending on its argument's type. Expansion of a functional unknown $f : a \to b$ depends on a:

- If $a = \mathsf{Bool}$, $f \in \{\lambda x.\ \mathsf{if}\ x\ t_1\ t_2\}$ where $t_1, t_2 : b$ are fresh unknowns of type b that are deeper than f.
- If a is uninterpreted, f is $\lambda x.\ \mathsf{switch}(x, m)$ where m is a table mapping $(\mathsf{elt}_i(a))_{i=0\ldots}$ to fresh unknowns of type b (built lazily, in practice) and switch is otherwise similar to case.
- If a is a datatype, f is either a constant function $\lambda x.\ k_f$ where k_f is an unknown of type b or $\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ c_i(\overline{y}) \to k_i\ \overline{y} \mid \cdots\ \mathsf{end}$ where each k_i is a fresh unknown taking the corresponding constructor's arguments as parameters. The constant case is used to be able to build functions that only peek superficially at inputs — otherwise, all function descriptions would be infinite in the model. The choice between the two forms for f is performed by the SAT solver; the non-constant case might be blocked by depth constraints. If a is infinite, bounded completeness is lost immediately, as we cannot generate all functions $a \to b$.
- Otherwise, a is a function type and we should reject the initial problem.

7 Refinements to the Calculus

The calculus can be improved with some refinements and strategies. These will only affect the concrete behavior of an implementation, not the scope of the language it accepts.

7.1 Multiple Conflict Clauses

Sometimes, a partial model causes a failure for several reasons: in the presence of parallel conjunction, both formulas can reduce to $\bot$. It would be wasteful to keep only one reason, because all of them might be useful to prune other branches. In this case, instead of just picking one explanation and discard the others, as suggested in Figure 2, we add a new explanation constructor, $e_1 \oplus e_2$, that combines two unrelated explanations, such that $\oplus$ is associative and commutative and $(e_1 \oplus e_2) \cup e_3 \equiv (e_1 \cup e_3) \oplus (e_2 \cup e_3)$. Intuitively, $a \xrightarrow{\rho}_{e_1 \oplus e_2} b$ means that a evaluates to b under substitution ρ assuming the choices in e_1 or in e_2 are made — those choices are never incompatible, but they might not be the same subset of $\mathsf{subst}(M)$. We add a new rule for $\wedge$:

$$\frac{a \xrightarrow{\rho}{}^*_{e_a} \bot \qquad b \xrightarrow{\rho}{}^*_{e_b} \bot}{a \wedge b \xrightarrow{\rho}_{e_a \oplus e_b} \bot}\ \textbf{and-left-right}$$

In case of conflict $\bigwedge_{t \in G} \xrightarrow{\rho}{}^*_{\bigoplus_{i \in I} e_i} \bot$, we obtain a set of conflict clauses $\left\{\bigvee_{a \in e_i} \neg a \mid i \in I\right\}$ that will prune distinct parts of the partial model.

7.2 Unification Rules

Equality already has many rules, but we can optimize it further. In our implementation, relying on hash-consing, we simplify $t = t$ into $\top$ in constant

time, even when t contains unassigned unknowns. We can optimize equality further in the special case where reduction leads to a term $c(t_1, \ldots, t_n) = k$ or $k = c(t_1, \ldots, t_n)$ where k is an unknown and c a constructor or domain element. This term reduces with no explanation to $\mathsf{if}\ \mathsf{check}_{[\![k\ :=c(u_1,\ldots,u_n)]\!]}\ (\bigwedge_{i=1}^{n} t_i = u_i)\ \bot$, where $\mathsf{check}_{[\![k\ :=c(u_1,\ldots,u_n)]\!]} : \mathsf{Bool}$ is a new term construct that requires $c(u_1, \ldots, u_n)$ to be one of the cases resulting from the expansion of k. In the $\bot$ case, the explanation forces the SAT solver to pick $c(u_1, \ldots, u_n)$ instead of ruling out the wrong choice $d(u_1, \ldots, u_m)$; if there are more than two constructors, this forces directly the right choice instead of trying every wrong choice.

$$\frac{c(u_1, \ldots, u_n) \text{ is a case of } k}{k = c(t_1, \ldots, t_n) \xrightarrow{\rho}_{\emptyset} \mathsf{if}\ \mathsf{check}_{[\![k\ :=c(u_1,\ldots,u_n)]\!]}\ (\bigwedge_{i=1}^{n} t_i = u_i)\ \bot}\ \textbf{unify}$$

$$\frac{\rho(k) = c(u_1, \ldots, u_n)}{\mathsf{check}_{[\![k\ :=c(u_1,\ldots,u_n)]\!]} \xrightarrow{\rho}_{\{k:=c(u_1,\ldots,u_n)\}} \top}\ \textbf{check-true}$$

$$\frac{\rho(k) = d(u_1, \ldots, u_m) \qquad d \neq c}{\mathsf{check}_{[\![k\ :=c(u_1,\ldots,u_n)]\!]} \xrightarrow{\rho}_{\{\neg(k:=c(u_1,\ldots,u_n))\}} \bot}\ \textbf{check-false}$$

7.3 Skewed Depth

Depth with a uniform cost for all constructors is a very coarse-grained measure of how simple a given input is, in particular when there are recursive datatypes with some high-arity constructors. Using constructors with several arguments typically tend to increase the size of the search space quickly. Therefore, it makes sense to penalize their use by using the flexible notion of depth $\mathsf{depth}(c(t_1, \ldots, t_n)) = \mathsf{cost}(c) + \max_{i=1\ldots n} \mathsf{depth}(t_i)$ with $\mathsf{cost}(c) = \mathrm{arity}(c)$. Many variations on this theme are possible.

7.4 Replacing Depth by Size

Instead of limiting the depth (or cost) of the unknowns, one could draw inspiration from Feat [11] and limiting their *size* in the iterative process of Algorithm 1. The promise of size-based limitation is that with datatypes that have constructors of high arity, some values of hugely different sizes have the same depth; when looking for small counter-examples it therefore seems better to favor small values rather than shallow ones. This idea is not implemented in our prototype, and is non trivial to write in an efficient way. The reason is that the computation of the size of an unknown (a property that requires looking at its entire shape) must be delegated to the SAT solver, whereas depth- or cost-based limitations can be constrained by only looking at each unknown individually.

7.5 Per-Type Cardinality Limit

Not every unknown or type will need grow at the same pace. For example, on a problem that involves an interpreter for untyped λ-calculus, limited by

some *fuel* (isomorphic to Nat) to ensure termination, it is clear that unknowns of functional types should be explored slowly, while initial fuel should grow fast enough to evaluate even the simplest terms. Similarly, in a problem mixing regular expressions (where depth gets costly very quickly) and Peano numbers, variables of type Nat should be allowed to get deeper than variables of type regex.

To this end, we can simply have one *family* of depth-limiting literals per type, of the form $[\![\mathsf{depth}_\tau \leq d]\!]$ for each type τ. At a given time, we maintain a mapping from types to depth limits, and only increase the depth of those types τ such that $[\![\mathsf{depth}_\tau \leq d]\!]$ occurs in the unsat-core. This refinement is not implemented in SMBC. This idea is similar to the incremental solving scheme in Paradox [7], except that we would limit the *maximal cardinal* of uninterpreted types and unknowns instead of directly setting their cardinal.

TODO$_{(1410)}$ change this, not exact

8 Implementation

We implemented SMBC in OCaml[5] using a modular SAT solver[6] that is flexible enough that we can add clauses dynamically and parameterize it with a theory solver. It also supports incremental solving under assumptions, which is necessary for the efficiency of the iterative deepening exploration. The core solver is around 3,200 lines long, including the term data structures, the symbolic evaluation and the main loop. This implementation is a prototype that can be used, but we believe it could be made much faster with more work and perhaps by using a lower-level language. The code is free software, under a permissive license.

Our description of evaluation rules in Figure 2 is quite high-level and can be implemented in various ways.[7] We chose to represent terms as perfectly shared directed acyclic graphs in which binders and bound variables rely on De Bruijn indices. The perfect sharing diminishes memory usage and makes let statements superfluous. We store in every term a pointer to a pair (explanation, term) that stores the current normal form of this term, effectively implementing a crude form of memoization. Any assignment of this pair must be undone upon backtracking — in a similar way as in congruence closure algorithms [3]. Similarly, unknowns are records with mutable pointers to a list of possible cases (once they have been expanded) and to their current assignment, which is reverted during backjumping thanks to a central backtracking stack that is controlled by the SAT solver. A good representation of explanations is required for efficiency, because union will be performed very often during the evaluation of terms and should be as fast as possible.

In addition, the evaluation function performs acyclicity checks to prune impossible branches early, and aggressively caches the normal forms of terms, stash-

[5] `https://github.com/c-cube/smbc/`

[6] `https://github.com/Gbury/mSAT`

[7] For example, it might be possible to write an efficient interpreter or compiler for use-cases where evaluation is the bottleneck, as long as explanations are tracked accurately and parallel conjunction is accounted for.

ing their old value on the central backtracking stack. Since we follow the architecture proposed by Barrett et al. [3], SMBC can delegate all branching to the SAT solver. Every time a boolean decision is made by the SAT solver (followed by propagation), the evaluation engine is called so as to prune bad models early. It does so by re-evaluating the set of goals G, which must contain at least one term not reduced yet, and cannot contain $\bot$ (see Algorithm 2). This re-evaluation is made faster by starting from the cached normal forms instead of the original goals. If all goals in G reduce to $\top$, the model is valid; if one of them reduces to $\bot$, the SAT solver immediately receives a conflict clause that will make it backtrack.

9 Experiments

We ran a few experiments to compare SMBC with other approaches, namely LSC, HBMC, CVC4 [18], and Inox, a standalone version of Leon [20]. We do not compare against QuickCheck, SmallCheck, or Feat, because they are not designed to solve such tightly constrained problems. All the data and the code of SMBC can be found at `https://cedeela.fr/~simon/files/cade_17.tar.gz`. For this experiment, we wrote some problems and borrowed some others from HBMC's test suite. We tried to pick diversified benchmarks so as to expose the strengths and weaknesses of each tool. TIP does not come yet with an exhaustive set of satisfiable benchmarks that would rely primarily on recursive functions. Benchmarks from our previous work on CVC4 [18] are expressed in SMT-LIB rather than TIP and use quantified axioms instead of recursive definitions, which makes them hard to use in our purely computational setting. The same holds of SMT-LIB and TPTP in general.

The solvers were run on a 4-cores Intel i5 CPU with 60 seconds timeout and a limit of 8 GB of RAM. Below, we give some numbers in Table 1 and then analyse the results on some categories of problems. The second column of the table is the number of satisfiable and unsatisfiable problems. Categories out of scope are marked with "–".

Problems	(SAT–UNSAT)	SMBC	HBMC	LSC	CVC4	Inox
Expr	(3–1)	2–0	**3–0**	2–0	0–0	**3–0**
Fold	(2–0)	**2–0**	–	–	–	–
Palindromes	(1–2)	**1–2**	1–1	0–0	0–0	0–1
Pigeon	(0–1)	**0–1**	–	–	**0–1**	0–0
Regex	(12–0)	7–0	2–0	**11–0**	–	0–0
Sorted	(2–2)	**2–2**	**2–2**	2–0	0–1	2–1
Sudoku	(1–0)	**1–0**	**1–0**	0–0	0–0	0–0
Type Checking	(2–0)	**2–0**	**2–0**	0–0	0–0	0–0

Table 1. Results of the Experiments

Expr Given arithmetic expressions, an evaluation function and several flawed simplifications, the goal is to find an expression such that its simplification does not evaluate to the same term. Here HBMC and Inox shine, but SMBC and LSC have more trouble due to the large branching factor of the search tree.

Fold Those examples are about synthesizing a function that distinguishes between lists by only looking at one element at a time (plus an accumulator). In other words, we fold a function f on all elements, and the goal is to pick f such that it can distinguish between close, but distinct, lists. This problem is outside the scope of all the tools the author knows about, simply because it combines an uninterpreted type with an unknown of function type, but SMBC has no problem synthesizing what is in essence a state machine transition function.

Palindromes After defining unary natural numbers and lists, we look for lists that are palindromes (i.e., $\mathsf{rev}(l) = l$) that have some additional constraint on their sum or length. Some of those problems are more difficult variations of the problem from Section 5.1. Some of the problems are satisfiable and some are unsatisfiable. For example, the goal in `long_rev_sum2.smt2` is to disprove the existence of a palindrome of length 200 with sum 1; HBMC times out because there are too many computations, and LSC cannot detect unsatisfiability. Those problems are easy but the toplevel goal is a parallel conjunction that needs to be treated properly, which is why LSC fails to solve even the satisfiable instances.

Pigeon A computational version of the classical pigeon hole problem, here with 4 holes for 5 pigeons. This requires handling uninterpreted types and unsatisfiable problems.

Regex Basic regular expressions are represented by a datatype featuring constants, star, and disjunction. The goal is generally to find a regular expression that matches a given string. Here, LSC shines and HBMC is in trouble, because (comparatively) many computations are required to check each input. SMBC has a good success rate here, even though its relatively naive interpreter is much slower than LSC's native compiled code.

Sorted Some problems about finding sorted lists of natural numbers that have additional properties (on their length, reverse, sum, etc.). The problems are fairly easy, but some of them are unsatisfiable.

Sudoku A sudoku is represented as a list of lists of a datatype with 9 constructors. Some functions to check whether the sudoku is valid (no duplicate in any line, column or block) are defined, an initial state is given, and the goal is simply to solve the sudoku. This is also a combinatorial problem on which HBMC takes only 2 s, SMBC takes 12 s, and LSC times out. Here, it pays to bit-blast because the SAT solver can propagate constraints among the sudoku cells.

Type Checking This example comes from the HBMC draft report [8]. Terms of the simply typed λ-calculus are defined by a datatype (variables being mapped to De Bruijn indices), along with a type-checking function that takes a term t, a type τ and an environment Γ (i.e. a list of types), and

returns $\top$ iff $\Gamma \vdash t : \tau$ holds. The goal is to find a term that has type $(a \to b) \to (b \to c) \to (a \to c)$ in the empty environment: in other words, to synthesize the composition operator from its type. The task is difficult because of the fast growth of the search space, in which LSC drowns, but SMBC manages well.

Overall, SMBC appears to be well balanced and to have good results both on problems that require computations and on problems where pruning of impossible cases is critical. Given the simplicity of our implementation, we believe these results are promising, and that SMBC occupies a sweet spot between handling computations well and traversing the search space in a smart way.

10 Perspectives for SMT solvers

SMBC builds on symbolic evaluation of terms in a SMT-like architecture where evaluation (propagation) and boolean reasoning are interleaved. It would be interesting to integrate it within existing SMT techniques: datatypes and recursive functions would be handled by computations, but integer or bitvector constraints would be managed by the dedicated decision procedures.

Such an integration would be beneficial to SMT solvers. SMT solvers are usually efficient on ground terms and ground axioms, as long as the theories are known beforehand and hard-coded into decision procedures. However, when the user wishes to use their own theories, their only path forward is to use quantified axioms; this usually results in a loss of completeness, as it is difficult to find a model for a quantified formula and check its validity. On the other hand, terminating recursive functions have a canonical model, as they are more restricted then arbitrary quantified formulas. Many theories should be expressible purely in terms of functions and datatypes: for example , the theory of (intensional) arrays would correspond to the following specification:

```
(declare-datatypes (a b)
  ((array (Const (unconst b)) (Set (key a) (value b) (tail array)))))

(define-fun-rec
 (par (a b)
   (get ((k a) (arr (array a b))) b
    (match arr
     (case (Const x) x)
     (case (Set k2 v2 arr2)
      (ite (= k k2) v2 (get k arr2)))))))
```

Similarly, one could specify many theories of data-structures, such as lists, balanced trees, functional queues, etc. without ever using a quantifier. Additionally, reasoning about functional program would be straightforward; reasoning on imperative programs would be possible after translating loops to recursive functions (perhaps with an additional decreasing parameter to enforce termination).

11 Conclusion

After describing a new technique for finding models in a logic of computable functions and datatypes, we presented ways of extending the language and described a working implementation. By combining symbolic evaluation with SAT-based conflict analysis, the approach is aimed at difficult problems where the search space is large (e.g., because of parallel disjunction and independent sub-problems) and large amounts of computations must be performed before discovering failure. It can be described as a spiritual heir to evaluation-driven narrowing [15] that replaces traditional exploration of the space of possible inputs by conflict driven clause learning. We hope that this work will benefit model finders in proof assistants, in particular Nunchaku [10, 18].

Acknowledgments The author would like to thank Jasmin Blanchette, Martin Brain, Raphaël Cauderlier, Koen Claessen, Pascal Fontaine, Andrew Reynolds, and Martin Riener, and the anonymous reviewers, for discussing details of this work and suggesting textual improvements.

References

1. Sergio Antoy, Rachid Echahed, and Michael Hanus. A Needed Narrowing Strategy. *Journal of the ACM (JACM)*, 2000.
2. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard Version 2.6. `http://www.SMT-LIB.org`, 2016.
3. Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 512–526. Springer, 2006.
4. Clark Barrett, Igor Shikanian, and Cesare Tinelli. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.
5. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
6. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of Inductive Problems. In *Conferences on Intelligent Computer Mathematics*, pages 333–337. Springer, 2015.
7. Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27, 2003.
8. Claessen, Koen and Rosén, Dan. SAT-based Bounded Model Checking for Functional Programs. `https://github.com/danr/hbmc` (unpublished), 2016.
9. The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr/`.
10. Simon Cruanes and Jasmin Christian Blanchette. Extending Nunchaku to Dependent Type Theory. In Jasmin Christian Blanchette and Cezary Kaliszyk, editors, *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016.*, volume 210 of *EPTCS*, pages 3–12, 2016.
11. Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional Enumeration of Algebraic Types. *ACM SIGPLAN Notices*, 47(12):61–72, 2013.

12. Michael Hanus. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.
13. Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96*, pages 23–34. IEEE, 1996.
14. Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.
15. Fredrik Lindblad. Property directed generation of first-order test data. In *Trends in Functional Programming*, pages 105–123. Citeseer, 2007.
16. Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.
17. Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co) datatypes in SMT solvers. In *International Conference on Automated Deduction*, pages 197–213. Springer, 2015.
18. Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model Finding for Recursive Functions in SMT. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2016.
19. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.
20. Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *International Static Analysis Symposium*, pages 298–315. Springer, 2011.