Satisfiability Modulo Bounded Checking

Simon Cruanes

Veridis, Inria Nancy
https://cedeela.fr/~simon/

CADE, August, 2017

Summary

Model Finding in a Computational Logic

SMBC = Narrowing + SAT

Extensions

Implementation & Experiments

Context

goal

- find counter-examples (typically, for proof assistants/testing)
- logic = recursive functions + datatypes

Example

Ask the solver to find a palindrome list of length 2 (e.g. [1;1]).

```
let rec length = function
  | [] -> 0
  | _ :: tail -> succ (length tail)

let rec rev = function
  | [] -> []
  | x :: tail -> rev tail @ [x]

(* magic happens here *)
goal exists (l:nat list), (rev l = l && length l = 2)
```

More examples

Example

Ask the solver to find a regex matching "aabb"

```
type char = A | B
type string = char list
type regex =
  | Epsilon (* empty *)
  | Char of char
  | Star of regex
  | Or of regex * regex (* choice *)
  | Concat of regex * regex (* concatenation *)

let rec match_re : regex -> string -> bool = ...

goal exists (r:regex), (match_re r [A;A;B;B])
```

We get `r = (ϵ|a*)·b*`, i.e.

r = Concat (Or (Epsilon, (Star (Char A))), Star (Char B))

More examples

Example

Solving a sudoku

```
type cell = C1 | C2 | ... | C9
type 'a sudoku = 'a list list

let rec is_instance : cell sudoku -> cell option sudoku -> bool = (* ... *)

let rec is_valid : cell sudoku -> bool = (* ... *)

(* the initial state, with holes *)
let partial_sudoku : cell option sudoku = [[None; Some C1; ...]; ...; ]

(* find a full sudoku that matches "partial_sudoku" *)
goal exists (e:cell sudoku), (is_instance e partial_sudoku && is_valid e)
```

→ combinatorial explosion, large search space (9^{81} grids)

→ we can solve in 14 s (with a general-purpose tool!)

More examples

Example

Simply-typed lambda calculus + typechecker (checks $\Gamma \vdash t : \tau$)

```
type ty = A | B | C | Arrow of ty * ty
type expr = Var of nat | App of expr * expr * ty | Lam of expr
type env = ty option list

let rec find_env : env -> nat -> ty option =
  (* ... *)

let rec type_check : env -> expr -> ty -> bool =
  (* ... *)

(* find e of type "(a -> b) -> (b -> c) -> (a -> c)" *)
goal exists (e:expr),
  (type_check [] e
    (Arrow (Arrow A B) (Arrow (Arrow B C) (Arrow A C))))
```

Logic: Recursion + Datatypes

- recursive datatypes: nat, list, tree, etc.
- simply typed
- recursive functions, assumed to be:

total	:	defined on every input
terminating	:	must terminate on every input

- **meta-variables**: undefined constants of some type
- boolean connectives, equality
- (optional) uninterpreted types, with finite quantification
- (optional) higher-order functions

We want to find a model: map each *meta-variable* to a concrete term built from constructors

Similar Tools

- **HBMC** : source of inspiration and examples and examples, bit-blasting Haskell $\rightarrow$ SAT
- **SmallCheck** : native code, tries all values up to depth k
- **Lazy SmallCheck** : same, but uses lazyness to expand
- **Leon** : lazy expansion of function calls in Z3
- **narrowing** : similar to LSC, refine meta-variables on demand
- **CVC4** : handles datatypes and recursive functions by quantifier instantiation + finite model finding ($\rightarrow$ less efficient?)
- **QuickCheck & co** : random generation of inputs. Very bad on tight constraints.
- **AFL-fuzz** : program instrumentation + genetic programming to evolve inputs.
- ...

Draw inspiration from HBMC / narrowing+SAT.

Summary

Model Finding in a Computational Logic

SMBC = Narrowing + SAT

Extensions

Implementation & Experiments

Narrowing

- lazy symbolic evaluation: expressions can evaluate to
  1. *blocked* expressions (**if** a b c blocked by a, etc.)
  2. normal forms (starts with true/false/constructor). Actually WHNF
- use *parallel and* (a && b reduce as soon as $a = \textbf{false} \lor b = \textbf{false} \lor (a = \textbf{true} \land b = \textbf{true})$)
- search loop:
  1. let $M := \emptyset$
  2. evaluate goal g symbolically in M
     - if g=**true**, success
     - if g=**false**, failure (backtrack/prune)
     - otherwise it must be blocked by $\{c_1, \dots, c_n\}$ (meta-vars)
  3. pick $c \in \{c_1, \dots, c_n\}$, expand it (e.g. $c = 0 \lor c = succ(c'))$)
  4. branch over the possible cases for c (e.g., let $M := M \cup \{c := 0\}$)
  5. goto step 3

Narrowing: an Example

goal (rev l1 @ rev l2 != rev (l1 @ l2))

- pick l1 = nil
- goal $\longrightarrow$ **false**
- backtrack; pick l1 = cons(x,l1')
- goal $\longrightarrow$ (rev l1' @ [x]) @ rev l2 != rev (l1'@l2) @ [x]
- pick l1' = nil
- goal $\longrightarrow$ ([x] @ rev l2 @ != rev l2 @ [x]
- pick l2 = nil, fail
- backtrack; pick l2=cons(y, l2')
- goal $\longrightarrow$ (x :: (rev l2' @ [y]) @ != (rev l2' @ [y]) @ [x]
- pick l2' = nil
- goal $\longrightarrow$ [x,y] != [y,x] $\longrightarrow$ x != y
- pick x=0, y=s(y')
- goal $\longrightarrow$ **true**: success!

Problem with Narrowing

$\boxed{\textbf{goal}\ \text{exists (l:nat list), (length l = 2 \&\& sum l = 500 \&\& rev l = l)}}$

Problem with Narrowing

goal exists (l:nat list), (length l = 2 && sum l = 500 && rev l = l)

Consider this execution

- pick l = cons(x,l')
- pick x = succ(x2)
- pick x2 = succ(x3)
- ...
- pick x500 = 0 (so that sum l=500)
- pick l' = nil
- (length l=2) $\longrightarrow$ **false**

$\rightarrow$ failure! But unrelated to choices on $x_{251}, x_{252}, \ldots, x_{500}$

$\rightarrow$ (wrong) choices on *shape* $\neq$ choices on *content*

$\rightarrow$ should have found l=cons(x1,cons(x2,nil)) early so rev(l)=l could prune this earlier

Problem with Narrowing (continued)

Consider:

```
type atom = True | False
type form = At of atom | And of form * form | Not of form

let imply a b = Not (And (a, Not b))

let rec eval (f:form): bool = match f with
  | At True -> true | At false -> false
  | Not f -> not (eval f)
  | And (a,b) -> eval a && eval b

(* yields "unsat" *)
goal not (eval (imply (imply (At a) (At b)) (imply (Not (At b)) (Not (At a)))))
```

Problem with Narrowing (continued)

Consider:

```
type atom = True | False
type form = At of atom | And of form * form | Not of form

let imply a b = Not (And (a, Not b))

let rec eval (f:form): bool = match f with
  | At True -> true | At false -> false
  | Not f -> not (eval f)
  | And (a,b) -> eval a && eval b

(* yields "unsat" *)
goal not (eval (imply (imply (At a) (At b)) (imply (Not (At b)) (Not (At a)))))
```

We obtain a basic SAT solver by just defining the evaluation function!

But DFS/BFS enumeration of values is naive...

(useless example, but think of ternary, multivalued, ... logics)

How to fix it?

Idea: use a SAT solver for non-chronological backtracking

- SAT solvers face the same issue
- CDCL: technique for tackling it efficiently
  (clause learning + backjumping)

→ essentially, learn *why* the conflict happened
  (and prevent it from ever happening again)

How to fix it?

Idea: use a SAT solver for non-chronological backtracking

- SAT solvers face the same issue
- CDCL: technique for tackling it efficiently
  (clause learning + backjumping)

$\rightarrow$ essentially, learn *why* the conflict happened
(and prevent it from ever happening again)

SMBC

- same basics as narrowing
- let the SAT solver choose between
  $\llbracket l := nil \rrbracket \vee \llbracket l := cons(x, l') \rrbracket$
- keep track of explanations e why $t \longrightarrow_e t'$
- upon failure:
  - conflict: $g \longrightarrow_e$ **false**
  - assert $\bigvee_{\llbracket c := t \rrbracket \in e} \neg \llbracket c := t \rrbracket$ to SAT solver

Illustration

goal exists (l:nat list), (length l = 2 && sum l = 500 && rev l = l)

Previous execution

- pick l = cons(x,l')
- pick x = succ(x2)
- pick x2 = succ(x3)
- ...
- pick x500 = 0 (so that sum l=500)
- pick l' = nil
- (length l=2) $\longrightarrow_{\{\llbracket l{:=}\mathrm{cons}(x,l')\rrbracket,\llbracket l'{:=}\mathrm{nil}\rrbracket\}}$ **false**
- conflict clause: $\neg\llbracket l := \mathrm{cons}(x, l')\rrbracket \vee \neg\llbracket l' := \mathrm{nil}\rrbracket$

Soon, $l = \mathrm{cons}(x, \mathrm{cons}(x', \mathrm{nil}))$ is proved
Then, constraints on $x, x_1, x_2, \ldots$ are progressively refined

Evaluation Rules

$$\frac{a \longrightarrow_e \textbf{true}}{\mathit{if}\ \ a\ b\ c \longrightarrow_e b}\ \textbf{if-left} \qquad \frac{a \longrightarrow_e \textbf{false}}{\mathit{if}\ \ a\ b\ c \longrightarrow_e c}\ \textbf{if-right}$$

$$\frac{f \longrightarrow_e g}{f\ x \longrightarrow_e g\ x}\ \textbf{app} \qquad \frac{a \longrightarrow_{e_1} b \qquad b \longrightarrow_{e_2} c}{a \longrightarrow_{e_1 \cup e_2} c}\ \textbf{trans}$$

$$\frac{}{(\lambda x.\ t)\ u \longrightarrow_\emptyset t[x := u]}\ \beta \qquad \frac{x \equiv t}{x \longrightarrow_\emptyset t}\ \textbf{def}$$

$$\frac{\mathsf{x} := \mathsf{c}}{x \longrightarrow_{\{\mathsf{x}:=\mathsf{c}\}} \mathsf{c}}\ \textbf{decision} \qquad \frac{b \longrightarrow_e \textbf{false}}{a \wedge b \longrightarrow_e \textbf{false}}\ \textbf{and-right}$$

$$\frac{a \longrightarrow_{e_a} \textbf{true} \qquad b \longrightarrow_{e_b} \textbf{true}}{a \wedge b \longrightarrow_{e_a \cup e_b} \textbf{true}}\ \textbf{and-true}$$

(omitted: and-left, pattern matching)

$\rightarrow$ careful with explanations of parallel and

Main Loop

For $d \in \{1, 2, \ldots, \text{max depth}\}$:

1. add assumption $\llbracket \text{depth} \leq d \rrbracket$
2. let $M := \emptyset$ (in SAT solver)
3. evaluate goal g symbolically in M
   - if $g \longrightarrow_e$ **true**, return SAT
   - if $g \longrightarrow_e$ **false**, conflict:
     Add $\left(\bigvee_{a \in e} \neg a\right)$ to SAT-solver;
     go to step 3
   - otherwise it must be blocked by $\{c_1, \ldots, c_n\}$ (meta-vars)
4. pick $c \in \{c_1, \ldots, c_n\}$, expand it
   (e.g. $\llbracket c = 0 \rrbracket \oplus \llbracket c = \mathsf{S}(c') \rrbracket$)
5. SAT-solver updates M
6. go to step 3

(omitted: decide if "unsat" related to assumption)

Iterative Deepening

The SAT solver decides which path to take.
To ensure fairness, we use Iterative Deepening.

Example

```
type t = A | B | C of t
```

Refine $x : t$ (of depth n) with clauses:

$$\llbracket x := A \rrbracket \vee \llbracket x := B \rrbracket \vee \llbracket x := C(y) \rrbracket \quad (y : t \text{ fresh unknown})$$
$$\neg\llbracket x := A \rrbracket \vee \neg\llbracket x := B \rrbracket$$
$$\neg\llbracket x := A \rrbracket \vee \neg\llbracket x := C(y) \rrbracket$$
$$\neg\llbracket x := B \rrbracket \vee \neg\llbracket x := C(y) \rrbracket$$
$$\neg\llbracket x := C(y) \rrbracket \vee \neg\llbracket \text{depth} \leq n \rrbracket$$

- special bool literal $\llbracket \text{depth} \leq n \rrbracket$ for each $n \in \mathbb{N}^+$
- refining a variable (x : List) of depth n
  $\rightarrow$ guard recursive cases with $\neg\llbracket \text{depth} \leq n \rrbracket$
- solve under assumption $\llbracket \text{depth} \leq n \rrbracket$; if unsat increase limit

Summary

Uninterpreted Types

Uninterpreted type τ: mapped into finite domain

- type slices $\tau_{[0\ldots]}, \tau_{[1\ldots]}, \tau_{[2\ldots]}, \ldots$ (where $\tau \stackrel{\text{def}}{=} \tau_{[0\ldots]}$)
- $\tau_{[n\ldots]} \stackrel{\text{def}}{=} \{\mathsf{elt}_n(\tau), \ldots, \mathsf{elt}_{\mathsf{card}(\tau)-1}(\tau)\}$.
- literals $[\![\mathsf{empty}(\cdot)]\!]$
  - $[\![\mathsf{empty}(\tau)]\!]$ means $\tau_{[n\ldots]} \equiv \emptyset)$
  - $\neg[\![\mathsf{empty}(\tau)]\!]$ means $\tau_{[n\ldots]} \equiv \{\mathsf{elt}_n(\tau)\} \cup \tau_{[n+1\ldots]}$
  - assume $\neg[\![\mathsf{empty}(\tau_{[0\ldots]})]\!]$

Uninterpreted Types

Uninterpreted type τ: mapped into finite domain

- type slices $\tau_{[0\ldots]}, \tau_{[1\ldots]}, \tau_{[2\ldots]}, \ldots$ (where $\tau \stackrel{\text{def}}{=} \tau_{[0\ldots]}$)
- $\tau_{[n\ldots]} \stackrel{\text{def}}{=} \{\mathrm{elt}_n(\tau), \ldots, \mathrm{elt}_{\mathrm{card}(\tau)-1}(\tau)\}$.
- literals $\llbracket \mathrm{empty}(\cdot) \rrbracket$
  - $\llbracket \mathrm{empty}(\tau) \rrbracket$ means $\tau_{[n\ldots]} \equiv \emptyset$)
  - $\neg\llbracket \mathrm{empty}(\tau) \rrbracket$ means $\tau_{[n\ldots]} \equiv \{\mathrm{elt}_n(\tau)\} \cup \tau_{[n+1\ldots]}$
  - assume $\neg\llbracket \mathrm{empty}(\tau_{[0\ldots]}) \rrbracket$

Quantification on finite types is just finite (con|dis)junction:

$$\frac{\rho(\llbracket \mathrm{empty}(\tau_{[n\ldots]}) \rrbracket) = \top}{\left(\forall x : \tau_{[n\ldots]}.\ F\right) \longrightarrow_{\{\llbracket \mathrm{empty}(\tau_{[n\ldots]}) \rrbracket\}} \top}$$

$$\frac{\rho(\llbracket \mathrm{empty}(\tau_{[n\ldots]}) \rrbracket) = \bot}{\left(\forall x : \tau_{[n\ldots]}.\ F\right) \longrightarrow_{\{\neg\llbracket \mathrm{empty}(\tau_{[n\ldots]}) \rrbracket\}} \left(F[\mathrm{elt}_n(\tau)/x] \wedge \left(\forall x : \tau_{[n+1\ldots]}.\ F\right)\right)}$$

Higher-Order

- can expand an unknown $f : a \to b$ (incomplete expansion)
- depends on a!
  - if $a = \text{bool}$, $\boxed{f \mapsto \lambda x.\ \text{if } x\ f_1\ f_2}$
    (where $f_1, f_2 : b$ are fresh unknowns)
  - if a is a datatype (e.g. nat),
    $\boxed{f \mapsto \lambda x : a.\ \text{case } x \text{ of } p_1 \to \ldots \mid \ldots \mid p_n \to \ldots \text{ end}}$
  - if a uninterpreted type, one mapping per domain element
  - if a is itself a function... more complicated (apply it to a set of terms)

$\to$ can find functions that examine a shallow prefix of their arguments
(finite decision trees)

Summary

Model Finding in a Computational Logic

SMBC = Narrowing + SAT

Extensions

Implementation & Experiments

Implementation

- OCaml implementation (`https://github.com/c-cube/smbc/`)
- based on msat (functorized SAT solver)
- parses TIP formulas (close to SMT-Lib + rec functions)
- 3,200 loc for the core Solver
- mostly tested on a small set of examples so far
- optimizations:
  - cached normal forms (with backtracking + path compression)
  - hashconsing for sharing terms (and normal forms)
  - most critical part: **evaluation** (use De Bruijn indices)
  - explanation: unbalanced tree for fast union
- to check if depth-limit actually responsible for "unsat"
  - re-run SAT solver without the assumption
  - could also use Unsat-core

Some basic experiments

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

(mostly combinatorial problems; many thanks to Koen Claessen and Dan Rosén for advice and problem files)

Conclusion

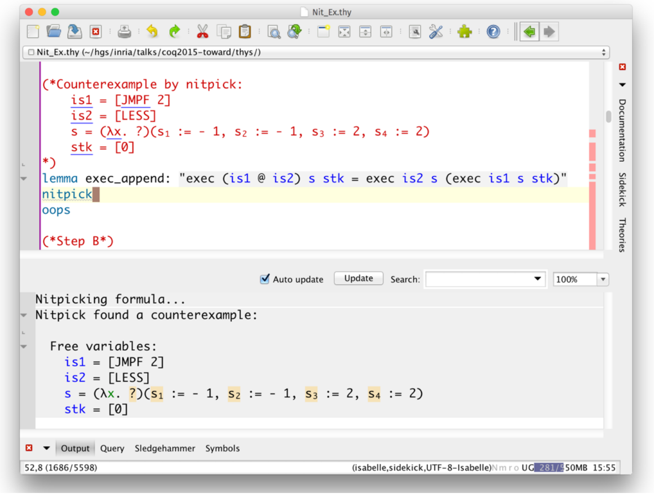
- computation+datatypes: **useful** and **widely applicable** problem!
- improve **narrowing** with **conflict-driven** backtracking: SAT-solvers know how to backtrack!
- current SMBC: straightforward implementation, but should be able to do better (maybe inside a SMT)
- many ideas of **improvement**
  - get closer to SMT (congruence closure, symbolic equality, etc.)
  - use rewriting rules/Horn clauses instead of functions+if+match
  - symmetry breaking, when constants play identical roles
- integration in Nunchaku (counter-model finder):
  - Coq/Lean frontend should yield many computational problems
  - CVC4, Kodkod, Paradox not very good on this fragment

  $\rightarrow$ complements well previous backends

Thank you for your attention!

Nunchaku: a successor to Nitpick

Nitpick: model finder integrated in Isabelle/HOL



Nunchaku in a Nu{t,n}shell

Design Decisions

- Decoupled from proof assistants
  - should be usable from several proof assistants
  - communicate via text and sub-processes
  - custom input/output language
- Decoupled from solvers
  - support several model finders/solvers as *backends*
  - CVC4, Paradox, Kodkod, SMBC
- Modular and maintainable

$\rightarrow$ modular pipeline of encoding/decoding passes (to each solver its own pipeline)

Obtaining a Model

Given this input:

```
val a : type.

codata llist := LNil | LCons a llist.

val xs : llist.

goal exists x. xs = LCons x xs.  # cyclic list
```

$ nunchaku problem.nun

Obtaining a Model

Given this input:

```
val a : type.

codata llist := LNil | LCons a llist.

val xs : llist.

goal exists x. xs = LCons x xs.  # cyclic list
```

$ nunchaku problem.nun

We obtain a finite model of a cyclic list $[a_1, a_1, \ldots]$

```
SAT: {
  type a := {$a_0, $a_1}.
  val xs := (mu (self_0/302:llist). LCons $a_1 self_0/302).
}
{backend:smbc, time:0.0s}
```

Encodings (translate input problem for solvers)

Bidirectional pipeline (composition of transformations)

forward: translate problem into simpler logic

backward: translate model back into original logic

Pipeline for SMBC (simplified)

1. type inference
2. monomorphization
3. compilation of multiple equations into pattern matching
4. specialization
5. elimination of codatatypes
6. polarization
7. elimination of inductive predicates
8. elimination of quantifiers on infinite types (functions, datatypes)
9. call SMBC

MainLoop

Require: Step ≥ 1: depth increment, G: set of goals

function MainLoop(G)
    $d \leftarrow$ Step ▷ initial depth
    while $d \leq$ MaxDepth **do**
        res $\leftarrow$ SolveUpTo(G, d)
        if res $=$ Sat **then return** Sat
        else if $\llbracket \text{depth} \leq d \rrbracket \notin$ UnsatCore(res) **then return** Unsat
        else $d \leftarrow d +$ Step
    return Unknown

SolveUpTo

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

```
function SolveUpTo(G, d)
    AddAssumption(⟦depth ≤ d⟧)
    M ∥ F ← ∅ ∥ G                                ▷ initial model and clauses
    while true do
        M ∥ F ← MakeSatDecision(M ∥ F )          ▷ model still partial
        M ∥ F ← BoolPropagate(M ∥ F)
        G′ ← {(u, e) | t ∈ G, t subst(M)*_e u}
        if (⊥, e) ∈ G′ then
            M ∥ F ← Conflict(M ∥ F ∪ {⋁_{a∈e} ¬a})
        else if all terms in G' are true  then return Sat
        else
            B ← ⋃_{(t,e)∈G′} block_{subst(M)}(t)    ▷ blocking unknowns
            for k ∈ B, k not expanded do
                F ← F ∪ Expand(k, d)             ▷ will add new literals
    and clauses
```

Expand

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

```
function Expand(k, d)
    let τ = c_1(τ_{1,1}, ..., τ_{1,n_1}) | ... | c_k(τ_{k,1}, ..., τ_{k,n_k})
    I ← {c_i(k_{i,1}, ..., k_{i,n_i}) | i ∈ 1, ..., k}        ▷ k_{i,j:τ_{i,j}} fresh meta
    AddSatClause(⋁_{t∈I} ⟦k := t⟧)
    AddSatClauses({¬⟦k := t_1⟧ ∨ ¬⟦k := t_2⟧ | (t_1, t_2) ∈ I, t_1 ≠ t_2})
    for t ∈ I where depth(t) > d do
        AddSatClause(¬⟦depth ≤ d⟧ ∨ ¬⟦k := t⟧)
```