SAT Modulo Bounded Checking

Simon Cruanes

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

22nd of June, 2017

Summary

1. Model Finding in a Computational Logic
2. SMBC = Narrowing + SAT
3. Current Limitations
4. Integration in Nunchaku

Example problem

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 (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 (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 = (* ... *)

let partial_sudoku : cell option sudoku = [[None; Some C1; ...]; ...; ]

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

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 = (* ... *)

let partial_sudoku : cell option sudoku = [[None; Some C1; ...]; ...; ]

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

→ combinatorial explosion, large search space

→ write a SMT solver (satisfiability modulo theory)

→ solves in 14 s (not bad for 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
  (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
- (optional) uninterpreted types, with finite quantification
- higher-order functions
- boolean connectives, equality

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

Similar Tools

HBMC : source of inspiration, bit-blasting Haskell $\to$ SAT

SmallCheck : native code, tries all values up to depth k

Lazy SmallCheck : same, but uses lazyness to expand

narrowing : similar to LSC, refine meta-variables on demand

CVC4 : handles datatypes and recursive functions by quantifier instantiation + finite model finding ($\to$ less efficient?)

QuickCheck & co : random generation of inputs. Very bad on tight constraints.

AFL-fuzz : program instrumentation + genetic programming to evolve inputs. Very IO-oriented.

...

Draw inspiration from HBMC / narrowing+SAT.

Summary

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 $\{c_1, \ldots, c_n\}$ (meta-vars)
  3. pick $c \in \{c_1, \ldots, c_n\}$, 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 2

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

goal (length l = 2 && sum l = 500)

Problem with Narrowing

goal (length l = 2 && sum l = 500)

Consider this execution:

- pick l = cons(x,l')
- pick x = succ(x2)
- ...
- pick x500 = 0 (for sum l=5)
- pick l' = nil
- length l=2 $\longrightarrow$ **false**

$\rightarrow$ failure! But all the choices on x, $x2$, ... are not related to this failure
$\rightarrow$ lots of useless backtracking

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

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

SMBC

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

Evaluation Rules

$$\frac{a \longrightarrow_e \mathbf{true}}{if\ \ a\ b\ c \longrightarrow_e b}\ \textbf{if-left}$$

$$\frac{a \longrightarrow_e \mathbf{false}}{if\ \ a\ b\ c \longrightarrow_e c}\ \textbf{if-right}$$

$$\frac{f \longrightarrow_e g}{f\ x \longrightarrow_e g\ x}\ \textbf{app}$$

$$\frac{a \longrightarrow_{e_1} b \qquad b \longrightarrow_{e_2} c}{a \longrightarrow_{e_1 \cup e_2} c}\ \textbf{trans}$$

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

$$\frac{x \equiv t}{x \longrightarrow_{\emptyset} t}\ \textbf{def}$$

$$\frac{\mathsf{x := c}}{x \longrightarrow_{\{\mathsf{x:=c}\}} \mathsf{c}}\ \textbf{decision}$$

$$\frac{b \longrightarrow_e \mathbf{false}}{a \wedge b \longrightarrow_e \mathbf{false}}\ \textbf{and-right}$$

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

(omitted: and-left, pattern matching)

$\rightarrow$ careful with explanations of parallel and

Implementation

- OCaml implementation (https://github.com/c-cube/smbc/)
- based on msat (functorized SAT solver)
- parses TIP formulas
- 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

Implementation

- OCaml implementation (`https://github.com/c-cube/smbc/`)
- based on msat (functorized SAT solver)
- parses TIP formulas
- 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
- iterative deepening for exploring bigger and bigger values
  - forbid branches that are too deep
  - if Unsat, check if depth limit in Unsat-core

Summary

1. Model Finding in a Computational Logic
2. SMBC = Narrowing + SAT
3. Current Limitations
4. Integration in Nunchaku

Issues

- even the type-driven synthesis is very hard
- need to prune branches faster
- need to reduce the size of the search space
- also missing: combination with theories? (arith, bitvectors, etc.)

What I tried

- unification rules (improves a bit: propagates)
- memoization (hard! Functions applied to *expressions*, return WHNF)
- delegate bool connectives to SAT (no change)

Issues

- even the type-driven synthesis is very hard
- need to prune branches faster
- need to reduce the size of the search space
- also missing: combination with theories? (arith, bitvectors, etc.)

What I tried

- unification rules (improves a bit: propagates)
- memoization (hard! Functions applied to *expressions*, return WHNF)
- delegate bool connectives to SAT (no change)

→ idea: more symbolic evaluation, more SMT

Idea: build on SMT

Just an idea, no implementation yet.

- use congruence-closure
  - allows efficient uninterpreted functions
  - allows (dis)unification rules
  - add notion of evaluation to it (**if**, **match**, β, …)
  - $\rightarrow$ merge congruence classes of a, b if $a \longrightarrow b$?
- let SAT solver pick WHNF of any term, not only meta-variables
- use datatype projectors (pred, head, tail) to deconstruct terms.
  Example: let $t \equiv$ **match** fact(n) **with** 0 −> a | succ x −> x+1

$$\frac{\text{is-succ(fact(n))}}{t \longrightarrow_{\{\text{is-succ(fact(n))}\}} \text{pred(fact(n))+1}}$$

  - need to check consistency between SAT decisions, and evaluation
  - more fine-grained tracking of reductions
  - additional reduction rule for projectors
- biggest issue: termination (depth-limit is more difficult)

Idea: use rewriting instead of pattern-matching

Example

```
type nat = Z | S of nat

let rec less (x:nat) (y:nat): bool = match x with
  | Z ->
    begin match y with
      | S _ -> true
      | Z -> false
    end
  | S x1 ->
    begin match y with
      | Z -> false
      | S y1 -> less x1 y1
    end
```

Idea: use rewriting instead of pattern-matching

Example

```
type nat = Z | S of nat

let rec less (x:nat) (y:nat): bool = match x with
  | Z ->
    begin match y with
      | S _ -> true
      | Z -> false
    end
  | S x1 ->
    begin match y with
      | Z -> false
      | S y1 -> less x1 y1
    end
```

Now, consider the expression less n Z (*n* blocked expression)

→ expression equivalent to **false**, but blocked!

→ need to refine *n*, growing search space for nothing

Idea: use rewriting instead of pattern-matching (cont'd)

Solution

Change language, use rewrite rules.

```
type nat = Z | S of nat

less _ Z          --> false.
less Z (S _)      --> true.
less (S x) (S y)  --> less x y.
```

Then, all rules whose LHS unifies with `less n Z` lead to **false**.

- close to *narrowing* as known in Term Rewriting
- can "match" on several arguments at a time
- SMBC would do a kind of E-unification (backed by SAT)

Missing: symmetry breaking?

- useful optimization in finite model finding
- if two meta $\{a, b\}$ "play the same role":
  1. pick one of them (say, a)
  2. add constraint $a \leq b$ for some (*builtin*) total order $\leq$

  $\rightarrow$ prunes the cases where $a > b$; no loss of completeness, by symmetry
- problem: not trivial here, every datatype value is distinct and can potentially be compared to any other

Perspective for SMT solvers

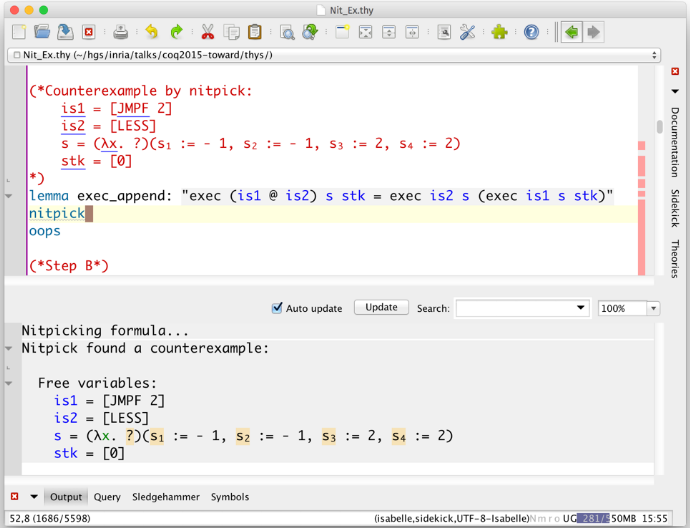
notion of evaluation very useful:

- can replace some (many?) uses of quantifiers with recursive functions
  - quantifiers are *bad*
  - can hope to find models (if types are finite)
- deal with **if** natively (no preprocessing)
- theory of arrays would just be one datatype + 2 functions!
- theories of data-structures: user-definable in many cases
- can talk about many programs directly:
  recursive functions, (bounded) loops, conditionals. . .
- simplifications (BV, arith) could be done on-the-fly

Summary

Nunchaku: a successor to Nitpick

Nitpick: model finder integrated in Isabelle/HOL



Design Decisions

- Decoupled from proof assistants
  - should be usable from serveral 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)

Input Language

```
data nat := Z | S nat.
data term :=
  | Var nat
  | Lam term
  | App term term.

rec bump : nat -> (nat -> term) -> nat -> term :=
  forall n rho. bump n rho Z = Var n;
  forall n rho m. bump n rho (S m) = bump (S n) rho m.

rec subst : (nat -> term) -> term -> term :=
  forall rho j. subst rho (Var j) = rho j;
  forall rho t. subst rho (Lam t) = Lam (subst (bump (S Z) rho) t);
  forall rho t u. subst rho (App t u) = App (subst rho t) (subst rho u).

goal exists t rho. subst rho t != t.
```

→ ML-like typed higher-order syntax

Here, find non-closed term t and subst ρ capturing a var of t

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.
```

$ 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.
```

$ 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

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

Conclusion

- improve narrowing with conflict-driven backtracking:
  SAT-solvers know how to backtrack!
- paper accepted at CADE
- current SMBC: decent implementation, but should be able to do better
- many ideas of improvement
  - get closer to SMT (congruence closure, symbolic equality, etc.)
  - use rewriting rules instead of functions+if+match
  - symmetry breaking, when constants play identical roles
- integration in Nunchaku:
  - Coq/Lean frontend should yield many computational problems
  - CVC4,paradox,Kodkod not very good on this fragment

  $\rightarrow$ complements well previous backends
- useful and widely applicable problem!

Thank you for your attention!