SMBC: Engineering a Fast Solver in OCaml

Simon Cruanes

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

28th of March, 2017

Summary

1. Presentation of SMBC (“Satisfiability Modulo Bounded Checking”)
2. Implementation
3. Profiling for Better Performance

Example problem

Given:

- inductive type declarations
- (recursive) function definitions
- a goal: an expression with variables in it

$\rightarrow$ find assignment of variables satisfying the goal

Example problem

Given:

- inductive type declarations
- (recursive) function definitions
- a goal: an expression with variables in it

→ find assignment of variables satisfying the goal

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)

Similar Tools

- HBMC : source of inspiration, bit-blasting Haskell $\rightarrow$ SAT
- small check : native code, tries all values up to depth k
- lazy small check : 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 ($\rightarrow$ inefficient?)
- QuickCheck & co : random generation of inputs. Very bad on tight constraints.
- ...

Draw inspiration from HBMC / narrowing+SAT.

The Bigger Picture

- make a better solver for problems based on recursive functions
- the rest of the talk: implementation
- use SMT techniques for not drowning in search space
  $\rightarrow$ "Satisfiability Modulo Bounded Checking"
- relation to Nunchaku (model finder for HO logic):
  - SMBC is a backend
  - other backends not very good on this fragment

$\rightarrow$ useful and widely applicable problem!

- paper submitted to CADE

Summary

1. Presentation of SMBC ("Satisfiability Modulo Bounded Checking")
2. Implementation
3. Profiling for Better Performance

Using a SAT-solver Library

Use a SAT solver, here .
→ does the backtracking and exploration.

Modularity

MSat is a *library* using an OCaml functor.
Bring your own theory!

```
module type THEORY = sig
  type formula
  type clause = (formula * bool) list

  type result = Ok of clause list | Conflict of clause

  val assume : formula -> bool -> result (* my code *)
end

module Sat(T:THEORY) : sig
  val solve : formula list -> bool (* library code *)
end
```

Internals

- one big Solver module (3,500 loc)
- more than 20 mutually recursive types at the beginning
  (to replace tables by embedding data inside objects)

→ yes, *ignoring* the rules of SW engineering can be fine

  ... for performance reasons

- one can write C in any language, even OCaml!

A few design Decisions

- terms (“expressions”, that is, trees) are DAGs with perfect sharing
  $\rightarrow$ save memory, constant-time comparison
  $\rightarrow$ most provers/SMT do it, even in C
- interpreter with caching of normal form of a value
  $\rightarrow$ caching often dramatically improves performance (when it applies)
- backtracking: a big stack of “undo” functions
  $\rightarrow$ low memory footprint, low overhead

Summary

1. Presentation of SMBC ("Satisfiability Modulo Bounded Checking")
2. Implementation
3. Profiling for Better Performance

How to debug Performance

SMBC needs to be very efficient, because it is *kind of* bruteforce.

Rules of thumb

- try to use efficient algorithms everywhere
- try to avoid allocating too much
- use compiler optimizations (here, `ocaml+flambda`)
- avoid obviously inefficient code, but:
- avoid "premature optimization" (as would say D. Knuth)

→ hence the need for profiling

CPU profiling: “perf”

perf: standard tool on Linux (initially there for the kernel)

CPU profiling: “perf”

perf: standard tool on Linux (initially there for the kernel)

```
% perf record --call-graph=dwarf ./smbc.native examples/ty_infer.smt2
(result SAT
 :model ((val e_2
          (lam
            (lam
              (lam
                (app (lam (app (var (s (s (s z)))) (var z) b))
                  (app (var (s z)) (var z) a) b)))))))
[ perf record: Woken up 40 times to write data ]
[ perf record: Captured and wrote 9.946 MB perf.data (1234 samples) ]

% perf report
```

Perf (cont'd)

```
Samples: 1K of event 'cycles:u', Event count (approx.): 1069569901
  Children      Self  Command      Shared Object       Symbol
+   99.12%     0.00%  smbc.native  smbc.native         [.] caml_main
+   99.12%     0.00%  smbc.native  smbc.native         [.] main
+   99.12%     0.00%  smbc.native  libc-2.24.so        [.] __libc_start_main
+   99.12%     0.00%  smbc.native  smbc.native         [.] _start
+   99.05%     0.00%  smbc.native  smbc.native         [.] camlSmbc__entry
+   99.05%     0.00%  smbc.native  smbc.native         [.] caml_program
+   99.05%     0.00%  smbc.native  smbc.native         [.] caml_start_program
+   98.98%     0.00%  smbc.native  smbc.native         [.] camlSmbc__solve_1377
+   98.94%     0.04%  smbc.native  smbc.native         [.] camlSolver__iter_7227
+   98.84%     0.00%  smbc.native  smbc.native         [.] camlMsat__External__solve_inner_4075
-   98.75%     1.50%  smbc.native  smbc.native         [.] camlMsat__Internal__search_1827
   - 97.25% camlMsat__Internal__search_1827
      + 46.61% camlMsat__Internal__theory_propagate_1801
      + 21.72% camlMsat__Internal__pick_branch_lit_1815
      + 13.89% camlMsat__Internal__propagate_atom_1763
      + 13.46% camlMsat__Internal__propagate_1802
      + 1.49% camlMsat__Internal__add_boolean_conflict_1716
   + 1.50% _start
+   46.61%     0.25%  smbc.native  smbc.native         [.] camlMsat__Internal__theory_propagate_1801
+   45.95%     0.08%  smbc.native  smbc.native         [.] camlSolver__assume_5801
+   41.55%     0.08%  smbc.native  smbc.native         [.] camlList__iter_1252
+   40.98%     0.16%  smbc.native  smbc.native         [.] camlSolver__update_5637
+   35.51%     0.58%  smbc.native  smbc.native         [.] camlSolver__compute_nf_add_5401
+   35.43%     1.90%  smbc.native  smbc.native         [.] camlSolver__compute_nf_noncached_5399
+   34.79%     2.41%  smbc.native  smbc.native         [.] camlSolver__compute_builtin_5402
+   21.72%     2.40%  smbc.native  smbc.native         [.] camlMsat__Internal__pick_branch_lit_1815
-   18.91%     4.39%  smbc.native  smbc.native         [.] camlMsat__Iheap__remove_min_1307
   - 14.53% camlMsat__Iheap__remove_min_1307
      + 10.93% camlMsat__Internal__f_weight_1510
      + 2.12% camlPervasives__max_1030
        1.07% caml_modify
   + 4.39% _start
+   17.92%     0.00%  smbc.native  smbc.native         [.] camlSolver__compute_nf_app_5400
```

Perf and Flamegraphs

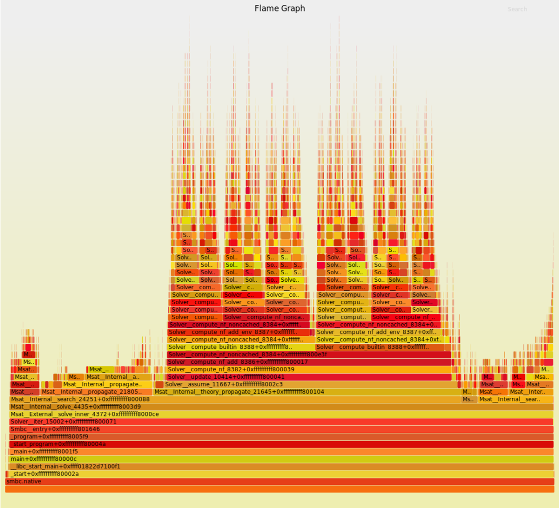
With deep recursive calls, `perf report` isn't very good.

Perf and Flamegraphs

With deep recursive calls, perf report isn't very good.

→ flame graphs (http://www.brendangregg.com/flamegraphs.html)

```
perf script \
  | stackcollapse−perf −−kernel \
  | sed 's/caml//g' \
  | flamegraph > perf.svg
```



Memory Profiling

OCaml has a GC, so I need to minimize allocations.

→ use spacetime, a new memory profiler!

- https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html
- https://blogs.janestreet.com/a-brief-trip-through-spacetime/

Memory Profiling

OCaml has a GC, so I need to minimize allocations.

→ use spacetime, a new memory profiler!

- https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html
- https://blogs.janestreet.com/a-brief-trip-through-spacetime/

Available as an OCaml *compiler switch*

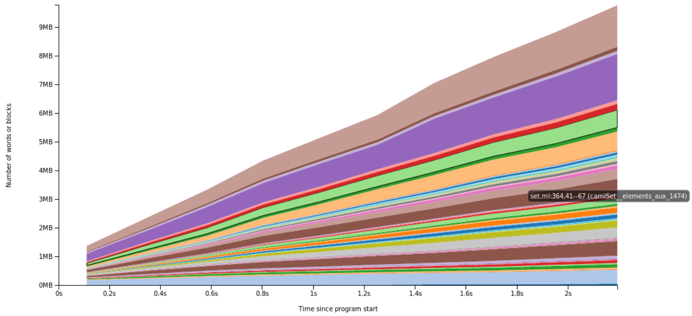
```
% opam sw 4.04.0+spacetime

% make clean all

% OCAML_SPACETIME_INTERVAL=100 ./smbc.native examples/ty_infer.smt2

% prof_spacetime serve spacetime-<PID> -e smbc.native
```

Memory Profiling (cont'd)

smbc.native

Mouse over the graph to show where values were allocated. Values allocated from non-OCaml code have their mouse-over popup text in green. Click a portion of the graph to move up the stack.

Live words   [Live blocks]   [All allocated words]



Backtrace (oldest frame first):

- (top of stack)

Horizontal: time

Vertical: space

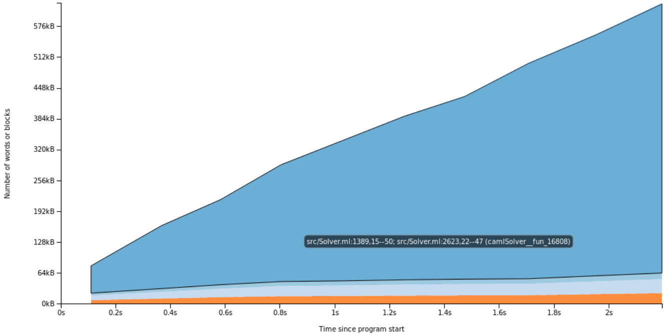
Colors: track memory allocated from a given program position

Memory Profiling (cont'd)

smbc.native

Mouse over the graph to show where values were allocated. Values allocated from non-OCaml code have their mouse-over popup text in green. Click a portion of the graph to move up the stack.

Live words   [Live blocks]   [All allocated words]



Backtrace (oldest frame first):

- set.ml:364,41--67 (camlSet__elements_aux_1474)
- (top of stack)

Can zoom into any region!

Memory Profiling

Found a performance bug this way:

- MSat uses a lot of dynamic arrays
- mistake in criterion for re-sizing

$\rightarrow$ resize at every `Vec.push`!

- almost all allocations came from there; spacetime made it obvious.

Conclusion

- good algorithms (SAT solver here) trump excellent implementation

. . . but implementation still important!

- OCaml can have reasonable performance if used properly
- profile before micro-optimizing
- tooling for profiling is tremendously useful

perf: can be used with many languages, de-facto standard on Linux

spacetime: awesome, but limited to OCaml

others: can also profile by manually inserting counters