Engineering Nunchaku: A Modular Pipeline of Codecs

Simon Cruanes

28th of June, 2016

Summary

1. Presentation of Nunchaku
2. Architecture Highlight: The Pipeline
3. Development Workflow and Lessons
4. Technical Issues so Far

Logic and Proof Assistants

Formal Logic: the science of accurate (deductive) reasoning

- operates on logic formulas
- precise notion of proof

Example

The very classic Socratic syllogism:

$$\frac{\text{man}(x) \Rightarrow \text{mortal}(x) \qquad \text{man}(\text{Socrates})}{\text{mortal}(\text{Socrates})}$$

$\rightarrow$ in practice, real proofs are difficult to handle

Proof Assistants

Hence, proof assistants

- manage the proof structure
- checks proofs (more trust)
- proof in the large (modularity, etc.)
- several competing logics and implementations

Coq	4 colors theorem, CompCert (developed at Inria)
Isabelle/HOL	SEL4 (TUM/Cambridge/. . .)
HOL light	Kepler Conjecture
. . .	other more "exotic" tools

- **large-scale proofs**: still a research topic (4 colors theorem, SEL4, Kepler conjecture. . .)

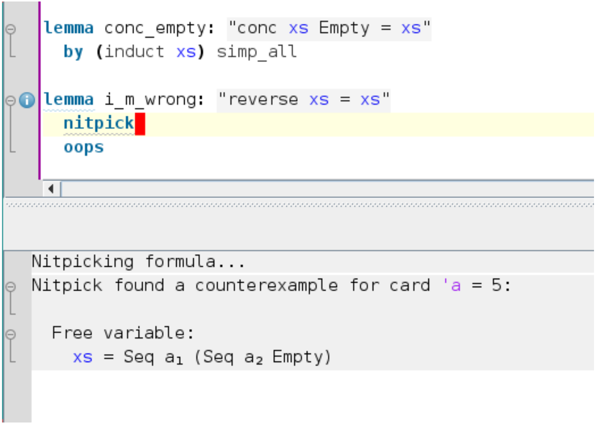
Isabelle/HOL

```
theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse :: "'a seq ⇒ 'a seq"
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all

lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"
  by (induct xs) simp_all

lemma reverse_conc: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
  by (induct xs) (simp_all add: conc_empty conc_assoc)

lemma reverse_reverse: "reverse (reverse xs) = xs"
  by (induct xs) (simp_all add: reverse_conc)
```

```
consts
  reverse :: "'a seq ⇒ 'a seq"
Found termination order: "size <*mlex*> {}"
```

Nitpick: Finding Mistakes

Tool integrated in Isabelle/HOL.



Here, it finds $[a_1, a_2]$ as a counter-example.

The genesis of Nunchaku

Issues with Nitpick

- hard to maintain (according to Jasmin)
- deeply tied to Isabelle (shared structures, poly/ML, ...)
- single logic backend (Kodkod)
  $\rightarrow$ we want to leverage modern research on SMT (CVC4)

The genesis of Nunchaku

Issues with Nitpick

- hard to maintain (according to Jasmin)
- deeply tied to Isabelle (shared structures, poly/ML, ...)
- single logic backend (Kodkod)

  → we want to leverage modern research on SMT (CVC4)

Project Nunchaku

- 2 years ADT (1 engineer)
- goal: useful for proof assistant users (not pure research prototype)
- standalone tool in OCaml

  → clean architecture, focus on maintainability and correctness

  → rich input language for expressing problems
- support multiple frontends (Isabelle, Coq, TLA+, ...)
- support multiple backends (CVC4, kodkod, HBMC?, ...)
- stronger/more accurate encodings (research part)

Nunchaku: the software

- free software (BSD license)
- uses git for versioning
- on the Inria forge and github
- modular OCaml code base
  - one transformation = one module
  - most functionality in a library, independent of CLI tool
  - dead simple input parser
- few dependencies
- communication with backends via **text** and subprocesses

Summary

1 Presentation of Nunchaku

2 Architecture Highlight: The Pipeline

3 Development Workflow and Lessons

4 Technical Issues so Far

The Big Picture

Nunchaku input/output

input : a logic problem (i.e. a formula from Isabelle)

output : (maybe) a counter-example in the same syntax

The Big Picture

Nunchaku input/output

input : a logic problem (i.e. a formula from Isabelle)

output : (maybe) a counter-example in the same syntax

Architecture

Bidirectional pipeline (composition of codecs)

forward: translate problem into simpler logic

backward: translate counter-example back into original logic

codec: a pair (encoder, decoder)

Similar to composition of passes in a compiler, *except* we also decode.

Codecs

Each codec as a pair of function

encode: `'a -> 'b * 'st` (encode + return some state)

decode: `'st -> 'c -> 'd` (decode using previously returned state)

```
type ('a, 'b, 'c, 'd, 'st) transformation_inner = {
  name : string;
  encode : 'a -> ('b * 'st);
  decode : 'st -> 'c -> 'd;
  (* ... *)
}

type ('a, 'b, 'c, 'd) transformation =
  Ex : ('a, 'b, 'c, 'd, 'st) transformation_inner ->
  ('a, 'b, 'c, 'd) t
```

→ use advanced features of OCaml (GADTs)

The Pipeline

The pipeline is a composition of codecs. . . with extras

```
val id : ('a, 'a, 'b, 'b) pipe

val compose :
  ('a, 'b, 'e, 'f) transformation ->
  ('b, 'c, 'd, 'e) pipe ->
  ('a, 'c, 'd, 'f) pipe

val fork :
  ('a, 'b, 'c, 'd) pipe ->
  ('a, 'b, 'c, 'd) pipe ->
  ('a, 'b, 'c, 'd) pipe
```

The Pipeline

The pipeline is a composition of codecs. . . with extras

```
val id : ('a, 'a, 'b, 'b) pipe

val compose :
  ('a, 'b, 'e, 'f) transformation ->
  ('b, 'c, 'd, 'e) pipe ->
  ('a, 'c, 'd, 'f) pipe

val fork :
  ('a, 'b, 'c, 'd) pipe ->
  ('a, 'b, 'c, 'd) pipe ->
  ('a, 'b, 'c, 'd) pipe
```

This way, many pipelines for different backends can be built safely. Made possible by OCaml's type system.

Current Pipeline

```
$ nunchaku --print-pipeline
Pipeline: ty_infer ==
          skolem ==
          mono ==
          elim_infinite ==
          elim_copy ==
          elim_multi_eqns ==
          specialize ==
          polarize ==
          unroll ==
          skolem ==
          fork { elim_ind_pred ==
                 elim_match ==
                 elim_data ==
                 lambda_lift ==
                 elim_hof ==
                 elim_rec ==
                 intro_guards ==
                 elim_prop_args ==
                 elim_types ==
                 close {to_fo ==
                        elim_ite == conv_tptp == paradox == id}
               | elim_ind_pred ==
                 lambda_lift ==
                 elim_hof ==
                 elim_rec ==
                 elim_match ==
                 intro_guards ==
                 close {to_fo == flatten {cvc4 == id}}
               }
```

Current Pipeline (in the code)

```
Tr.ElimIndPreds.pipe ~print:(!print_elim_preds_ || !print_all_) ~check @@@
fork
  (
    Tr.ElimPatternMatch.pipe ~print:(!print_elim_match_ || !print_all_) ~check @@@
    Tr.ElimData.pipe ~print:(!print_elim_data_ || !print_all_) ~check @@@
    Tr.LambdaLift.pipe ~print:(!print_lambda_lift_ || !print_all_) ~check @@@
    Tr.ElimHOF.pipe ~print:(!print_elim_hof_ || !print_all_) ~check @@@
    Tr.ElimRecursion.pipe ~print:(!print_elim_recursion_ || !print_all_) ~check @@@
    Tr.IntroGuards.pipe ~print:(!print_intro_guards_ || !print_all_) ~check @@@
    Tr.Elim_prop_args.pipe ~print:(!print_elim_prop_args_ || !print_all_) ~check @@@
    Tr.ElimTypes.pipe ~print:(!print_elim_types_ || !print_all_) ~check @@@
    Tr.Model_rename.pipe_rename ~print:(!print_model_ || !print_all_) @@@
    close_task (
      Step_tofo.pipe ~print:!print_all_ () @@@
      Tr.Elim_ite.pipe ~print:(!print_elim_ite_ || !print_all_) @@@
      FO.pipe_tptp @@@
      paradox
    )
  )
  ( Tr.LambdaLift.pipe ~print:(!print_lambda_lift_ || !print_all_) ~check @@@
    Tr.ElimHOF.pipe ~print:(!print_elim_hof_ || !print_all_) ~check @@@
    Tr.ElimRecursion.pipe ~print:(!print_elim_recursion_ || !print_all_) ~check @@@
    Tr.ElimPatternMatch.pipe ~print:(!print_elim_match_ || !print_all_) ~check @@@
    Tr.IntroGuards.pipe ~print:(!print_intro_guards_ || !print_all_) ~check @@@
    Tr.Model_rename.pipe_rename ~print:(!print_model_ || !print_all_) @@@
    close_task (
      Step_tofo.pipe ~print:!print_all_ () @@@
      Transform.Pipe.flatten cvc4
    )
  )
```

A word on the AST

AST: abstract syntax tree

- critical data structure in a logic-processing tool
- biggest issue: representation of variables

```
type var = private (string * int)
val new_var : string -> var

type term =
  | Var of var
  | Const of string (* function symbol *)
  | App of term * term list
  | Forall of var * term

type 'a subst = (var, 'a) map
```

Any operation recursing under `Forall` will rename variable on the fly.

Summary

1. Presentation of Nunchaku
2. Architecture Highlight: The Pipeline
3. Development Workflow and Lessons
4. Technical Issues so Far

vim + several plugins, including merlin

→ code completion, type inference

```
(* compute the set of specializable arguments in each function of [defs] *)
let compute_specializable_args_def ~self (defs : (_,_) Stmt.rec_defs) =
  let state = Trav.state self in
  let ids =
    Stmt.defined_of_recs defs
    |> Sequence.map Stmt.id_of_defined
    |> ID.Set.of_seq
  in
  let cga = mk_cga_state ~env:(Trav.env self) ids in
  (* process each definition *)
  List.iter
    (fun def ->
       let id = def.Stmt.rec_defined.Stmt.defined_head in
       let n = record_calls_def cga id def in
       let bv = bv_of_callgraph cga.cga_graph id n in
       ID.Tbl.replace state.specializable_args id bv;
       Utils.debugf ~section 3 "@[<2>can specialize `@[%a : %a@]` on:@ @[%a@]@]"
         (fun k->
            let ty = def.Stmt.rec_defined.Stmt.defined_ty in
            k ID.print_full id P.print ty CCFormat.(array bool) bv);
    )
    defs;
  Utils.debugf ~section 5 "@[<2>call graph: @[%a@]@]"
    (fun k->k CallGraph.print cga.cga_graph);
  ()

env:(term, term) Env.t -> ID.Set.t -> 'a call_graph_analyze_state
```

Git

I use git extensively to version the code:

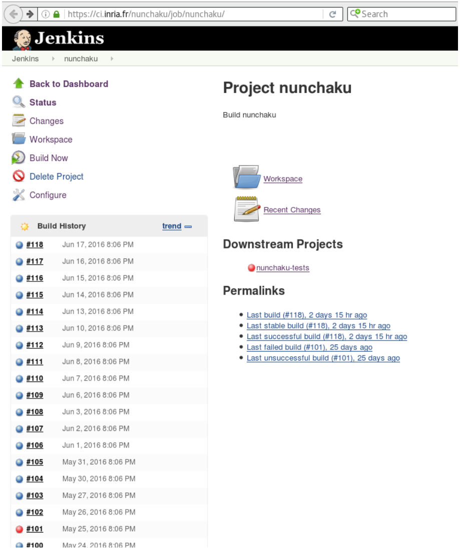
- *master* branch for stable versions
- *dev* for main development branch
- one feature = one branch
  1. develop in the branch
  2. when ready, merge in *dev*
  3. when dev "works" (passes tests), merge in *master*

Continuous Integration

Use of `https://ci.inria.fr` for:

- testing that the code builds
- running the test suite

→ plans for emitting Junit reports (report % of failed tests)



Testing

- almost no unit test: code depends on deeply nested data structures
- whole-program tests: a repository of test problems
  - problems contain their "expected" status
  - tool to run on the whole suite and check results
  - very useful for regressions
- wip: tighter integration with Jenkins (use Junit format for results)

Debugging

printf-like debugging

- use the `Format` module extensively
  - → nice pretty-printing module
  - → I write a printer for almost every type
- activated by command-line flags, *per-module*
- also able to print output of *each codec*

```
0.008[unif] unify `prop` and `prop`
0.008[ty_infer]
   generalize `~ cons a (cons b nil) = cons b (cons a nil)`, by forall
0.008[ty_infer]
   checked statement `goal ~ cons a (cons b nil) = cons b (cons a nil).`
after type inference: {
  rec choice : pi (a/4:type). a/4 -> prop -> a/4 :=
        forall p/8:(a_0/6 -> prop).
          choice a_0/6 p/8 =
          (choice a_0/6 p/8)
          asserting
            (p/8 = (fun (x/10:a_0/6). (false)) || p/8 (choice a_0/6 p/8)).
```

Handling Dependencies

Reinventing the wheel is bad.

- We use opam with a few dependencies
  - → a parser generator (menhir)
  - → a standard library extension
  - → basic threading and unix building blocks
- distribution as a binary, so far (static linking)
- be careful of non-portable dependencies

Summary

1. Presentation of Nunchaku
2. Architecture Highlight: The Pipeline
3. Development Workflow and Lessons
4. Technical Issues so Far

Unix is tricky

Calling sub-processes properly is not easy

- call `Unix.setsid` to prevent them from surviving the caller
- careful with FD duplication, deadlocks, etc.

```
let popen cmd ~f =
  Unix.setsid ();
  (* spawn subprocess *)
  let stdout, p_stdout = Unix.pipe () in
  let stderr, p_stderr = Unix.pipe () in
  let p_stdin, stdin = Unix.pipe () in
  List.iter Unix.set_close_on_exec [stdin; stdout; stderr];
  let stdout = Unix.in_channel_of_descr stdout in
  let stdin = Unix.out_channel_of_descr stdin in
  let pid = Unix.create_process
      "/bin/sh" [| "/bin/sh"; "-c"; cmd |]
      p_stdin p_stdout p_stderr in
  Unix.close p_stdout; Unix.close p_stdin; Unix.close p_stderr;
  let cleanup () = (* ... *) in
  try
    let x = f (stdin, stdout) in
    let _, res = Unix.waitpid [Unix.WUNTRACED] pid in
    let res = match res with
      | Unix.WEXITED i | Unix.WSTOPPED i | Unix.WSIGNALED i -> i
    in
    cleanup ();
    res
  with e ->
    cleanup ();
    raise e
```

Concurrency is tricky

We run several sub-processes at the same time
$\rightarrow$ need concurrency

- nightmare to debug
- risk of deadlock
- risk of race condition

Concurrency is tricky

We run several sub-processes at the same time
→ need concurrency

- nightmare to debug
- risk of deadlock
- risk of race condition

Solution: futures

```
module Fut : sig
  type 'a t (* eventually contains a 'a value *)

  val return : 'a -> 'a t (* immediate *)
  val make : (unit -> 'a) -> 'a t (* execute in new thread *)
  val map : ('a -> 'b) -> 'a t -> 'b t
  val flat_map : ('a -> 'b t) -> 'a t -> 'b t

  type 'a final_state =
    | Stopped
    | Done of 'a
    | Fail of exn

  val stop : _ t -> unit
  val is_done : _ t -> bool

  val get : 'a t -> 'a final_state (* blocking *)
end
```

Conclusion

After 9 months, project has made good progress!

Status

- pipeline to CVC4 is complete
- pipelines to Kodkod and Paradox: wip
- around 20,000 lines of OCaml so far

Learnt from Implementing

- OCaml is awesome (!!)
- proper use of types:
  - enforces good abstraction
  - prevent many, many mistakes
- proper use of modules to keep codecs independent