Nunchaku: Flexible Model Finding for Higher-Order Logic

Simon Cruanes, Jasmin Blanchette, Andrew Reynolds

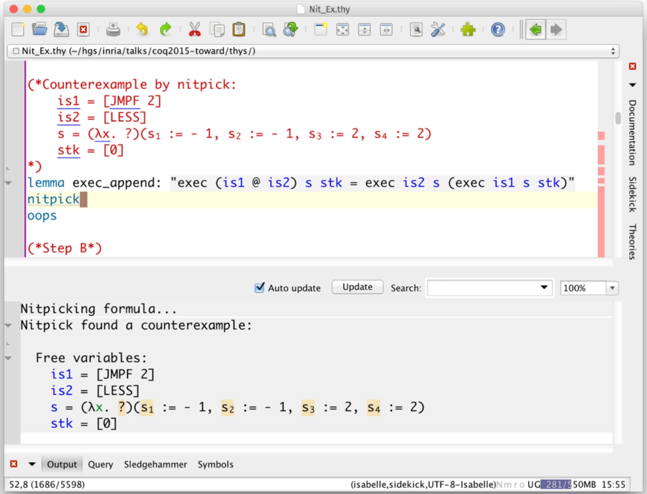
Veridis, Inria Nancy
https://cedeela.fr/~simon/

April 7th, 2016

Summary

Nitpick: the current titleholder

Tool integrated in Isabelle/HOL



Issues with Nitpick

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

The genesis of Nunchaku

Goals

- 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, probably kodkod, HBMC, ...)
- stronger/more accurate encodings

Who

- Jasmin Blanchette (lead, maintainer of Nitpick)
- Simon Cruanes (main developer)
- Andrew Reynolds (maintains finite-model finding in CVC4)

The genesis of Nunchaku

Goals

- useful for proof assistant users (not pure research prototype)
- standalone tool in OCaml
  - $\rightarrow$ clean architecture, focus on maintainability and correctness
  - $\rightarrow$ rich input language for expressing problems
- support multiple frontends (Isabelle, Coq, TLA+, ...)
- support multiple backends (CVC4, probably kodkod, HBMC, ...)
- stronger/more accurate encodings

Who

- Jasmin Blanchette (lead, maintainer of Nitpick)
- Simon Cruanes (main developer)
- Andrew Reynolds (maintains finite-model finding in CVC4)

Summary

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 ρ. bump n ρ Z = Var n;
  forall n ρ m. bump n ρ (S m) = bump (S n) ρ m.

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

goal exists t ρ. subst ρ t != t.
```

→ ML-like typed higher-order syntax

Here, find a non-closed term t and substitution ρ capturing one of its variables

Input Language: Codatatypes

```
val u : type.

codata tree :=
    | leaf u
    | node (llist u)
and llist a :=
    | lnil
    | lcons a (llist a).

goal exists x y z. x = node (lcons (leaf y) (lcons (leaf z) (lcons x lnil))).
```

Here, we ask Nunchaku to find the (infinite) tree
$\mu x.\text{node}\,[\text{leaf}\ y, \text{leaf}\ z, x]$ where y and z are arbitrary values of type u.

Input Logic

Features

- higher-order (with λ, partial application, etc.)
- polymorphic types
- (co)datatypes + pattern matching
- recursive definitions, axioms, (co)inductive predicates. . .
- extensionality, choice
- a type *prop* for formulas, with the usual connectives

Obtaining a Model

We obtain finite fragments of infinite models (after decoding)

Here: finite model of `even` and `odd` (= decision tree)

```
SAT: {
  val m := Succ Zero.
  val even :=
    fun (v_0/104 : nat).
      if v_0/104 = Succ (Succ Zero)
        then true
      else if v_0/104 = Zero
        then true
      else ?__ (even v_0/104).
  val odd :=
    fun (v_0/105 : nat).
      if v_0/105 = Succ Zero
        then true else ?__ (odd v_0/105).
}
```

The `?__` is an undefined value that is specific to this model.

Obtaining a Model

We obtain finite fragments of infinite models (after decoding)
Here: finite model of `even` and `odd` (= decision tree)

```
SAT: {
  val m := Succ Zero.
  val even :=
    fun (v_0/104 : nat).
      if v_0/104 = Succ (Succ Zero)
        then true
      else if v_0/104 = Zero
        then true
      else ?__ (even v_0/104).
  val odd :=
    fun (v_0/105 : nat).
      if v_0/105 = Succ Zero
        then true else ?__ (odd v_0/105).
}
```

The `?__` is an undefined value that is specific to this model.

Summary

The Big Picture

Bidirectional pipeline (composition of transformations)

forward: translate problem into simpler logic
backward: translate model back into original logic

Current Pipeline (simplified)

1. type inference
2. monomorphization
3. specialization
4. polarization
5. elimination of inductive predicates
6. elimination of higher-order functions
7. elimination of recursive functions
8. backend (CVC4)

The Big Picture

Bidirectional pipeline (composition of transformations)

forward: translate problem into simpler logic

backward: translate model back into original logic

Current Pipeline (simplified)

1. type inference
2. monomorphization
3. specialization
4. polarization
5. elimination of inductive predicates
6. elimination of higher-order functions
7. elimination of recursive functions
8. backend (CVC4)

Monomorphization

- from polymorphic logic to simply-typed logic
- only keep useful definitions
- instantiate polymorphic formulas, definitions, etc. with ground types
- incomplete in some cases (polymorphic recursion $\rightarrow$ depth limit)

Specialization

Specialize some higher-order functions on a static subset of their args
Goal: make the problem *less* higher-order

```
rec map : (a → a) → list → list :=
  forall f. map f nil = nil;
  forall f x l. map f (cons x l) = cons (f x) (map f l).
```

```
map (fun x. x + f y + z) l
```

We specialize `map` on its first argument:

```
rec map32 : a → a → list → list :=
  forall y z. map32 y z nil = nil;
  forall x y z l. map32 (cons x l) = cons (x + f y + z) (map32 y z l).
```

```
map32 y z l
```

(careful: need proper closure)

Specialization

Specialize some higher-order functions on a static subset of their args
Goal: make the problem *less* higher-order

```
rec map : (a → a) → list → list :=
  forall f. map f nil = nil;
  forall f x l. map f (cons x l) = cons (f x) (map f l).
```

```
map (fun x. x + f y + z) l
```

We specialize `map` on its first argument:

```
rec map32 : a → a → list → list :=
  forall y z. map32 y z nil = nil;
  forall x y z l. map32 (cons x l) = cons (x + f y + z) (map32 y z l).
```

```
map32 y z l
```

(careful: need proper closure)

Specialization

Specialize some higher-order functions on a static subset of their args
Goal: make the problem *less* higher-order

```
rec map : (a → a) → list → list :=
  forall f. map f nil = nil;
  forall f x l. map f (cons x l) = cons (f x) (map f l).
```

```
map (fun x. x + f y + z) l
```

We specialize `map` on its first argument:

```
rec map32 : a → a → list → list :=
  forall y z. map32 y z nil = nil;
  forall x y z l. map32 (cons x l) = cons (x + f y + z) (map32 y z l).
```

```
map32 y z l
```

(careful: need proper closure)

Polarization

Predicate p becomes p^+ and p^- (positive occurrences/negative occurrences).

- $\rightarrow$ in some case, gain precision
- $\rightarrow$ transform $\Leftrightarrow$ into $\Rightarrow$
- $\rightarrow$ more accurate Skolemization
- $\rightarrow$ unpolarized context (under $\Leftrightarrow$, say):
  $p^+ \ x$ asserting $p^+ \ x = p^- \ x$

Elimination of Inductive Predicates + Unrolling

```
pred odd : nat → prop :=
  odd (Succ Zero);
  forall n. odd n ⇒ odd n;  # not well-founded!
  forall n. odd n ⇒ odd (Succ (Succ n)).
```

becomes:

```
rec odd : nat → nat → prop :=
  forall decr n. odd decr n =
    exists pred_decr. decr = Succ pred_decr ∧
    (  n = Succ Zero
    ∨ odd pred_decr n
    ∨ exists m. n = Succ (Succ m) ∧ odd pred_decr m)
```

- `pred_decr`: unrolling (ensure WF)
- transform predicate into regular recursive function
- solver picks initial value of `decr`

Elimination of Inductive Predicates + Unrolling

```
pred odd : nat → prop :=
  odd (Succ Zero);
  forall n. odd n ⇒ odd n;  # not well-founded!
  forall n. odd n ⇒ odd (Succ (Succ n)).
```

becomes:

```
rec odd : nat → nat → prop :=
  forall decr n. odd decr n =
    exists pred_decr. decr = Succ pred_decr ∧
    (  n = Succ Zero
    ∨ odd pred_decr n
    ∨ exists m. n = Succ (Succ m) ∧ odd pred_decr m)
```

- `pred_decr`: unrolling (ensure WF)
- transform predicate into regular recursive function
- solver picks initial value of `decr`

Elimination of HOF

λ-lifting

Transform $C[\lambda x.\ x + g\ y + a]$ into $C[f_{37}\ y]$ where $f_{37}\ y\ x := x + g\ y + a$

Introduction of Application symbols

Replace $f : \tau_1 \rightarrow \tau_2$ by the constant f : to $\tau_1\ \tau_2$ and app_f : to $\tau_1\ \tau_2 \rightarrow \tau_1 \rightarrow \tau_2$

add guards and extensionality axioms to preserve semantics.

Elimination of HOF

λ-lifting

Transform $C[\lambda x.\ x + g\ y + a]$ into $C[f_{37}\ y]$ where
$f_{37}\ y\ x := x + g\ y + a$

Introduction of Application symbols

Replace $f : \tau_1 \to \tau_2$ by the constant $f : \text{to}\ \tau_1\ \tau_2$ and
$\text{app}_f : \text{to}\ \tau_1\ \tau_2 \to \tau_1 \to \tau_2$

add guards and extensionality axioms to preserve semantics.

Elimination of Recursive Functions

Recursive function $\forall x : \tau.\ f\ x := \ldots f\ y \ldots$

- intractable for finite model finding if $\text{card}(\tau) = \infty$
  - $\rightarrow$ quantify over α_τ, an unspecified finite subset of τ
  - $\rightarrow$ $\forall a : \alpha_\tau.\ f\ (\gamma_\tau a) := \ldots (f\ y \text{ asserting } \exists b : \alpha_\tau.\ y = \gamma_\tau\ b) \ldots$
- CVC4 supports quantification over finite types!
- transformation critical in practice (many recursive functions!)
- also work for productive functions on codata, tailrec functions. . .

(publications: SMT2015, IJCAR2016)

Backend (CVC4)

- send first-order problem to CVC4 (using SMTLIB syntax)
  - CVC4 supports finite model finding, (co)datatypes, ...
  - try different sets of options in parallel
- parse result (including model, if any)
- end of pipeline: easily replaced by another backend

We will soon add other backends!

$\rightarrow$ In some sense, Nunchaku is to become a unified language for model finders!

Summary

Nunchaku: the software

- free software (BSD license)
- 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

Transformations

Each transformation as a pair of function

- encode: $\alpha \to \beta * \texttt{'st}$ (encode + return some state)
- decode: $\texttt{'st} \to \gamma \to \delta$ (decode using previously returned state)

```
type (α, β, γ, δ, 'st) transformation_inner = {
  name : string;
  encode : α → (β * 'st);
  decode : 'st → γ → δ;
  (* ... *)
}

type (α, β, γ, δ) transformation =
  Ex : (α, β, γ, δ, 'st) transformation_inner → (α, β, γ, δ) t
```

Example

```
val monomorphization : (poly_pb, mono_pb, mono_model, poly_model) transformation
```

Transformations

Each transformation as a pair of function

encode: $\alpha \to \beta *$ 'st (encode + return some state)

decode: 'st $\to \gamma \to \delta$ (decode using previously returned state)

```
type (α, β, γ, δ, 'st) transformation_inner = {
  name : string;
  encode : α → (β * 'st);
  decode : 'st → γ → δ;
  (* ... *)
}

type (α, β, γ, δ) transformation =
  Ex : (α, β, γ, δ, 'st) transformation_inner → (α, β, γ, δ) t
```

Example

```
val monomorphization : (poly_pb, mono_pb, mono_model, poly_model) transformation
```

The Pipeline

The pipeline is a composition of transformations

```
val id : (α, α, β, β) pipe

val compose :
  (α, β, 'e, 'f) transformation →
  (β, γ, δ, 'e) pipe →
  (α, γ, δ, 'f) pipe
```

This way, many pipelines for different backends can be built safely, by mere composition.

The Pipeline

The pipeline is a composition of transformations

```
val id : (α, α, β, β) pipe

val compose :
  (α, β, 'e, 'f) transformation →
  (β, γ, δ, 'e) pipe →
  (α, γ, δ, 'f) pipe
```

This way, many pipelines for different backends can be built safely, by mere composition.

Summary

~~Future~~ Ongoing Work

Lots of things!

1. currently: finishing support for HOF
2. next: multi backends
3. dependent types in input? ($\rightarrow$ Coq, lean...)
4. more accurate translations
5. ...(research!)

Thank you for your attention!
Questions?

Model for the "De Bruijn" problem

```
SAT: {
val _witness_of
      (exists t/240:term (ρ/241:nat → term).
         ~ ((subst ρ/241 t/240) = t/240)) :=
  Var Z.
val bump :=
  fun (v_0/234 : nat) (v_1/235 : nat → term) (v_1/236 : nat).
    if v_0/234 = S Z ∧ v_1/235 = $to_nat_term_0 ∧ v_1/236 = Z
      then Var (S Z)
    else ?__ (bump v_0/234 v_1/235 v_1/236).
val subst :=
  fun (v_0/237 : nat → term) (v_1/238 : term).
    if v_0/237 = $to_nat_term_0 ∧ v_1/238 = Var Z
      then Var (S Z)
    else ?__ (subst v_0/237 v_1/238).
val _witness_of
      (exists (ρ/104:nat → term). ~ ((subst ρ/104 t/103) = t/103))
  :=
  fun (v_1/239 : nat).
    if v_1/239 = Z
      then Var (S Z) else ?__ (nun_sk_1 v_1/239).

}
```