Property-Based Testing for Coq

Cătălin Hrițcu

Prosecco Reading Group - Friday, November 29, 2013

The own itch I'm trying to scratch

- hard to devise correct safety and security enforcement mechanisms (static or dynamic)
  - type systems, reference monitors, ...
  - full confidence only with mechanized proofs
- frustrating to prove while designing mechanism
  - broken definitions and properties
  - countless iterations for ...
    - discovering the correct set of lemmas
    - strengthening inductive invariants
- other people might have similar itches

Dream

- Wouldn’t it be cool if Coq had a tactic for automagically producing counterexamples?
- Fortunately such tools already exist:
  - “The New Quickcheck for Isabelle” [Bulwahn, CPP 2012]
  - in fact, Isabelle has lots of push-button automation:
    - *proving*: Sledgehammer [Paulson et al, since approx 2006]
    - *disproving*: Quickcheck, Refute [Weber, ENTCS 2005], Nitpick [Blanchette & Nipkow, ITP2010]
- ... but nothing like this for Coq
  - Clear practical need: property-based testing for Coq
  - Question: Is there any interesting research left to do?

This talk

- **P**roperty-**B**ased **T**esting (PBT)
  - what it is, by example
  - the state of the art, quickly
- own experience with PBT
  - testing noninterference [ICFP 2013 and after]
  - prototype for random testing in Coq
- ideas for going beyond the state of the art
  - smart mutation testing
  - deep integration with Coq/SSReflect

PROPERTY-BASED TESTING

QuickCheck

[Claessen & Hughes, ICFP 2000]

- Property-based random testing for Haskell
  - **Demo**
- Using type classes for
  - type-based input generation and shrinking
- Probability is just a monad with random sampling as the action
- Highly customizable
  - write your own generators and shrinkers using reusable combinators (e.g. choose, frequency,...)

Custom generator (a simple one)

```
frequency $
  [ (1, pure Noop) ] ++
  [ (1, pure Halt) ] ++
  [ (10, pure Add) | nstk >= 2 ] ++
  [ (10, Push <$> lint) ] ++
  [ (10, pure Pop) | nstk >= 1 ] ++
  [ (20, pure Store) | nstk >= 2
                     , absAdjustAddr vtop `isIndex` mem ] ++
  [ (20, pure Load) | nstk >= 1
                    , absAdjustAddr vtop `isIndex` mem ] ++
  [ (10, liftM2 Call (choose (0, (nstk-1) `min` maxArgs)) arbitrary)
                      | nstk >= 1
                      , cally ] ++
  [ (20, liftM Return arbitrary) | Just r <- [ fmap astkReturns $
                                               find (not . isAData) stk]
                                 , nstk >= if r then 1 else 0
                                 , cally ] ++
  [ (10, pure Jump) | nstk >= 1
                    , jumpy ] ++
  [ (10, pure JumpNZ) | nstk >= 2
                      , genTMM ] ++
  [ (10, pure Sub) | nstk >= 2, genTMM ] ++
  [ (10, pure LabelOf) | labelOfAllowed $ gen_instrs getFlags ]
```

Input generation

- random is not the only way
  - **exhaustive testing with small instances**
    - SmallCheck for Haskell [Runciman et al, Haskell 2008]
    - New Quickcheck for Isabelle [Bulwahn, CPP 2012]
  - **symbolic / narrowing-based testing**
    - [Lindblad, TFP 2007]
    - EasyCheck for Curry [Christiansen & Fischer, FLOPS 2008]
    - Lazy SmallCheck for Haskell [Runciman et al, Haskell 2008]
    - New Quickcheck for Isabelle [Bulwahn, CPP 2012]
  - **constraint-programming-based**
    - FocalTest [Carlier et al. 2013]

Input generation (2)

- Smarter generation is not always better
  - generation time can dominate testing
- random generation
  - super customizable
    - precise probability distribution
    - often needs manual customization for good results
  - not predictable
    - that matters for proof scripts

Hitting sparse preconditions

- Trivial example: forall x y, x = y ==> P x y
- manually, using custom generator
  - choose (0, 10000)
  - $s_1 \approx s_2$ – generate s_1 then vary it to $s_2 \approx s_1$
- automatically
  - Glass-box testing of Curry programs [Fischer & Kuchen, PPDP 2007]
  - New Quickcheck for Isabelle [Bulwahn, LPAR 2012]
  - FocalTest [Carlier et al. 2013]

Executing declarative specifications (inductive definitions)

- again strong connection to functional logic programming
  - Mercury [Somogyi et all, since around 1994]
  - Curry [Hanus, POPL 1997]
- Isabelle/HOL
  - extraction, large TCB [Berghofer&Nipkow, TYPES 2002]
  - small TCB [Berghofer et al, TPHOLs 2009]
- Plugins for Coq producing ...
  - OCaml code, large TCB [Delahaye et al, TPHOLs 2007]
  - certified Coq code, small TCB [Tollitte et al, CPP 2012]

OWN EXPERIENCE WITH PBT

[ICFP 2013 and after]

Verifying security of the SAFE system

long term goal

- current status:
  noninterference in Coq for very simplified model
  [Azevedo de Amorim et al, POPL 2014]

- **However**…
  - Proofs for actual system **a lot more work**
  - Design is **still evolving**
  - **Feedback** on correctness needed ASAP

Random testing?

- Can we use QuickCheck for **noninterference**?

- **The experiment**
  - ***very simple*** **machine** (10 instructions)
  - **standard noninterference property**
  - generate many random programs
    and try to **find counterexamples**

Encouraging results

- introduced **plausible errors** in IFC rules
- **all errors found in 2-16ms** on average

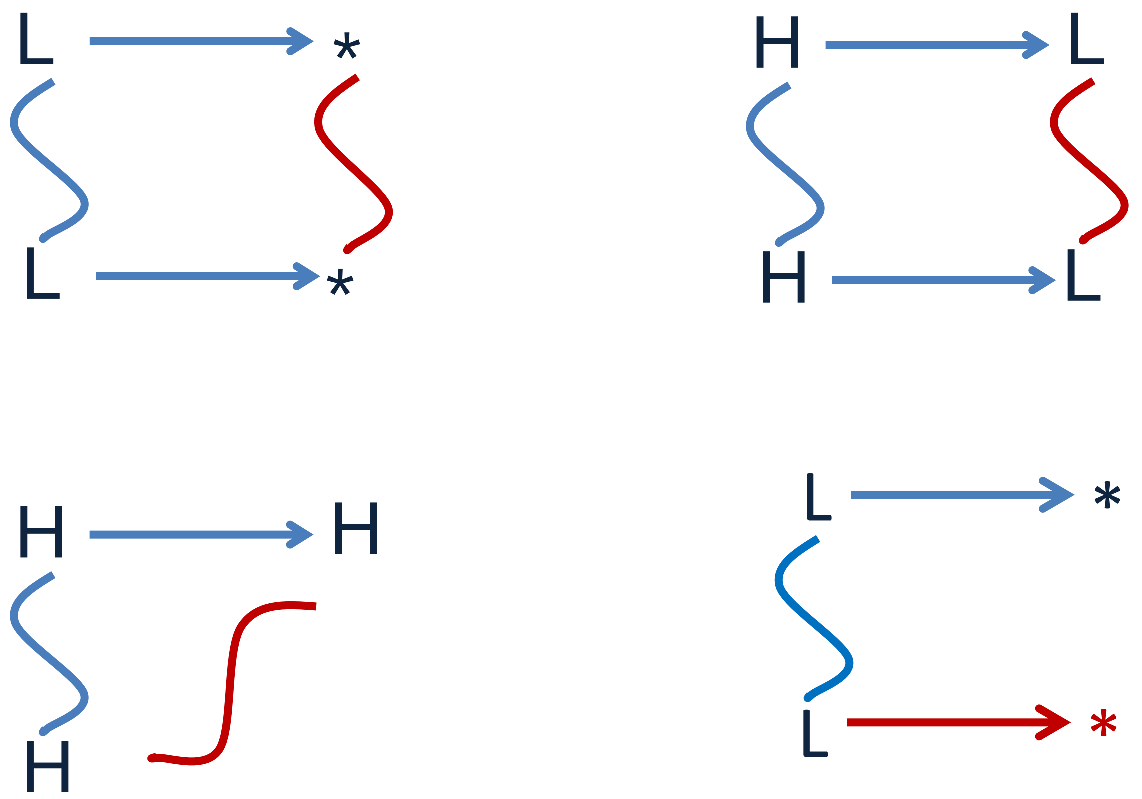
- **However**, for these results
  we are not using QuickCheck naïvely
  – that didn't really work for us
  – **significant cleverness was needed** in 3 areas…

The 3 secret ingredients

1. Clever **program generation strategies**
   – distributions, instruction sequences, smart integers
   – best one: “generation by execution”
2. **Strengthening the tested property**
   – best one: unwinding conditions (next slide)
   – requires inventing (by hand!) stronger invariants
      • invariants of real SAFE machine are very complicated
3. **Shrinking** counterexamples

Unwinding conditions

easiest to test and suitable for [co]inductive proof



QuickCheck clone for Coq (prototype)

- Initial testing noninterference work [ICFP 2013] used Haskell QuickCheck
- Since then Leo (Leonidas Lambropoulos) ported Haskell QuickCheck to Coq
- Using extraction only for
  – efficient evaluation, random seed, tracing
- **Demo**

Custom generator in Coq

```
frequency (pure Nop) [
  (* Nop *)
  (1, pure Nop);
  (* Halt *)
  (0, pure Halt);
  (* PcLab *)
  (10, liftGen PcLab genRegPtr);
  (* Lab *)
  (10, liftGen2 Lab genRegPtr genRegPtr);
  (* MLab *)
  (onNonEmpty dptr 10, liftGen2 MLab (elements Z0 dptr) genRegPtr);
  (* FlowsTo *)
  (onNonEmpty lab 10,
   liftGen3 FlowsTo (elements Z0 lab)
            (elements Z0 lab) genRegPtr);
  (* LJoin *)
  (onNonEmpty lab 10, liftGen3 LJoin (elements Z0 lab)
                              (elements Z0 lab) genRegPtr);
  (* PutBot *)
  (10, liftGen PutBot genRegPtr);
  (* BCall *)
  (10 * onNonEmpty cptr 1 * onNonEmpty lab 1,
   liftGen3 BCall (elements Z0 cptr) (elements Z0 lab) genRegPtr);
  (* BRet *)
  (if containsRet stk then 50 else 0, pure BRet);
  (* Alloc *)
  (200 * onNonEmpty num 1 * onNonEmpty lab 1,
   liftGen3 Alloc (elements Z0 num) (elements Z0 lab) genRegPtr);
```

IDEAS FOR EXTENDING THE STATE OF THE ART

- Smart Mutations
- Deep Integration with Coq/SSReflect

High confidence by PBT

- *"testing can only show the presence of bugs, not their absence"* – Dijkstra
- systematically introduce bugs and **test the testing infrastructure** (e.g. the generator)
  - if testing finds all introduced bugs but no new bugs then **we do get high confidence**
- initial experiments [ICFP 2013] added bugs manually
  - not good, turns code into spaghetti
- newer experiments with **smart mutation very encouraging**
  - can easily enumerate all missing taints and missing checks

Mutants game (input rule table)

	Allow	Result	PC
OpLab	TRUE	BOT	LabPC
OpMLab	TRUE	Lab1	LabPC
OpPcLab	TRUE	BOT	LabPC
OpBCall	TRUE	JOIN Lab2 LabPC	JOIN Lab1 LabPC
OpBRet	LE (JOIN Lab1 LabPC) (JOIN Lab2 Lab3)	Lab2	Lab3
OpFlowsTo	TRUE	JOIN Lab1 Lab2	LabPC
OpLJoin	TRUE	JOIN Lab1 Lab2	LabPC
OpPutBot	TRUE	BOT	LabPC
OpNop	TRUE	__	LabPC
OpPut	TRUE	BOT	LabPC
OpBinOp	TRUE	JOIN Lab1 Lab2	LabPC
OpJump	TRUE	__	JOIN LabPC Lab1
OpBNZ	TRUE	__	JOIN Lab1 LabPC
OpLoad	TRUE	Lab3	JOIN LabPC (JOIN Lab1 Lab2)
OpStore	LE (JOIN Lab1 LabPC) Lab2	Lab3	LabPC
...	...	...	...

Mutant game (final output)

```
./Extracted
Fighting 52 mutants
Killed mutant 0 (1 frags)
Killed mutant 1 (2 frags)
Killed mutant 2 (3 frags)
Killed mutant 3 (4 frags)
Killed mutant 4 (5 frags)
Killed mutant 5 (6 frags)
Killed mutant 6 (7 frags)
Killed mutant 7 (8 frags)
Killed mutant 8 (9 frags)
Killed mutant 9 (10 frags)
Killed mutant 10 (11 frags)
Killed mutant 11 (12 frags)
Killed mutant 12 (13 frags)
Killed mutant 13 (14 frags)
Killed mutant 14 (15 frags)
Killed mutant 15 (16 frags)
Killed mutant 16 (17 frags)
Killed mutant 17 (18 frags)
Killed mutant 18 (19 frags)
Killed mutant 19 (20 frags)
Killed mutant 20 (21 frags)
Killed mutant 21 (22 frags)
Killed mutant 22 (23 frags)
Killed mutant 23 (24 frags)
Killed mutant 24 (25 frags)
Killed mutant 25 (26 frags)
Killed mutant 26 (27 frags)
Killed mutant 27 (28 frags)
Killed mutant 28 (29 frags)
Killed mutant 29 (30 frags)
Killed mutant 30 (31 frags)
Killed mutant 31 (32 frags)
Killed mutant 32 (33 frags)
Killed mutant 33 (34 frags)
Killed mutant 34 (35 frags)
Killed mutant 35 (36 frags)
Killed mutant 36 (37 frags)
Killed mutant 37 (38 frags)
Missed mutant [38] (38 frags)
Missed mutant [39] (38 frags)
Killed mutant 40 (39 frags)
Killed mutant 41 (40 frags)
Killed mutant 42 (41 frags)
Killed mutant 43 (42 frags)
Killed mutant 44 (43 frags)
Killed mutant 45 (44 frags)
Killed mutant 46 (45 frags)
Killed mutant 47 (46 frags)
Killed mutant 48 (47 frags)
Killed mutant 49 (48 frags)
Killed mutant 50 (49 frags)
Killed mutant 51 (50 frags)
```

Iterative workflow

```
M,P,T := best guess for a mechanism, property, and test config
start:
if test(M,P,T) finds counter then
  (M := manual-fix M  ||  P := manual-fix P); goto start
else
  Ms := mutate M
  for each mutant Ms[i] do (even in parallel)
    if test(Ms[i],P,T) finds counter then
      killed[i] := true
    else
      killed[i] := false
      if manual-search(Ms[i],P) finds counter then
        T := manual-fix T; goto start
  if forall i we have killed[i] then
    done; validated P for M
  else
    for each j so that not(killed[j]) do
      M := apply change Ms[j] to M
    goto start
```

Smart mutation

- Mutation testing already exists
  - 390 papers from 1977 to 2009 [Jia & Harman, 2010]
  - TDD world: test suite = specification
    - any change in behavior that’s not caught by testing is considered a potential bug and manually inspected
    - kill count just another metric, an alternative to coverage
  - purely syntactic mutations
- Smart mutation not quite the same
  - PBT world: property = specification
  - only produce more permissive mechanism

Open problem

- Generalizing smart mutation beyond IFC, to arbitrary static or dynamic mechanisms
- very simple thing to try first:
  – dropping preconditions of inductive definition
  – making Boolean function return more true
  – can't do these properly in a black-box way; so even these require meta-programming

IDEAS FOR EXTENDING THE STATE OF THE ART

- Smart Mutations ✓
- Deep Integration with Coq/SSReflect

Testing actual lemmas / proof goals

- Currently
  - reimplement mechanism & property in the purely functional fragment of Coq
  - prove equivalence (or soundness?)
  - test this executable variant
- Ideally, switch freely between
  - proving and testing
  - declarative and executable ...

SSReflect

- in small-scale reflection proofs
  - defining both declarative and computational specs
  - switching freely between them

  ... is already the normal **proving** process
- testing would add small(er) additional overhead
- while SSReflect computational specifications are often not fully / efficiently executable
  - could use refinement framework by Denes et al. [ITP 2012, CPP 2013] for switching to efficiently executable specs

Potential workflow

- Reify proof goal to syntactic representation of formula (Coq plugin)
- Normalize formula (DNF, classically equivalent)
- Associate computations to atoms (type classes)
  - negative atoms (premises) get **smart generators**
    - optimization: smart generators only for *sparse* negative atoms
  - positive atoms (conclusions) get **checkers**
- Associate Skolem functions to existentials (type class)
- User would still have to provide type class instances
  - could try to use existing work for automating this

THANK YOU

Native Coq execution

- current prototype uses extraction
- want more seamless integration in Coq
  - make the result of testing and counterexamples available to Coq tactics and terms
- exploit recent progress on NativeCoq [Boespflug et al, CPP 2011]
  - this will only complement extraction
  - tracing for debugging still needs extraction

Prove things about generators

- Surjectivity [Dybjer et al, TPHOLs 2003]
- Correctness of smart generators

Testing with nondeterminism

- Oracles

Dependent types

- This is what Coq is all about