

Using Yices as an automated solver in Isabelle/HOL

Levent Erkök John Matthews
`{levent.erkok,matthews}@galois.com`

AFM'08: Automated Formal Methods 2008
Princeton, NJ

July 2008



- Providing strong assurance evidence for certification
 - Some properties are amenable for automated proof
 - For others, manual intervention is a must
- Strategy:
 - Use a theorem-proving framework
 - High-level correctness and “deeper” results
 - Aided by push-button techniques:
 - When the subgoal is sufficiently simple
... but usually very tedious ...
- Use whatever tool works the best
 - And combinations thereof

The ismt tactic

- We use Isabelle/HOL
 - Local expertise counts..
- The `ismt` tactic out-sources proofs to Yices
 - Directly supports a large chunk of HOL
 - Uses “uninterpretation” for the rest
- Similar to the `yices` strategy in PVS

- Proof-replay mode
 - Trust nothing; translate and replay the proof
 - High assurance; Runs slow and is expensive to build.

Modes of integration

- Proof-replay mode
 - Trust nothing; translate and replay the proof
 - High assurance; Runs slow and is expensive to build.
- Proof-check mode
 - Do not replay, but “validate” the proof
 - Medium (adjustable) assurance; Faster to run; Cheaper to build

Modes of integration

- Proof-replay mode
 - Trust nothing; translate and replay the proof
 - High assurance; Runs slow and is expensive to build.
- Proof-check mode
 - Do not replay, but “validate” the proof
 - Medium (adjustable) assurance; Faster to run; Cheaper to build
- Oracle mode
 - Trust everything!
 - Lowest assurance; Runs fast and cheapest to build
 - No proofs required from the external solver

Modes of integration

- Proof-replay mode
 - Trust nothing; translate and replay the proof
 - High assurance; Runs slow and is expensive to build.
- Proof-check mode
 - Do not replay, but “validate” the proof
 - Medium (adjustable) assurance; Faster to run; Cheaper to build
- Oracle mode
 - Trust everything!
 - Lowest assurance; Runs fast and cheapest to build
 - No proofs required from the external solver
- Proof generation for SMT solvers is still active research area
- Yices does not produce proofs; so oracle mode is the only choice

- 1 Introduction
- 2 Connecting Isabelle to Yices
- 3 Example Translations
- 4 Dealing with false alarms
- 5 Application: Verifying C programs
- 6 Summary

How does ismt work

- Grab the top-most goal from the Isabelle goal stack
- Translate the types involved to Yices
 - Might require “monomorphisation”
 - Introduce uninterpreted types as needed
- Negate the subgoal, and translate it to a Yices term
 - If no matching construct; uninterpret
- Pass the script to Yices
- If Yices decides the negation is unsatisfiable:
 - Trigger oracle mechanism to assert the goal proven
 - A “trust-tag” will be attached.

How does ismt work

- Grab the top-most goal from the Isabelle goal stack
- Translate the types involved to Yices
 - Might require “monomorphisation”
 - Introduce uninterpreted types as needed
- Negate the subgoal, and translate it to a Yices term
 - If no matching construct; uninterpret
- Pass the script to Yices
- If Yices decides the negation is unsatisfiable:
 - Trigger oracle mechanism to assert the goal proven
 - A “trust-tag” will be attached.
- What do we do if Yices returns a model?

Interpreting Yices's models

- Recall that the model is for the negation of the goal
- ..Hence, it is a counter-example to what we were trying to prove
- Typically indicates a bug found
- Models are translated back to Isabelle/HOL
 - Provides very valuable feedback!

Interpreting Yices's models

- Recall that the model is for the negation of the goal
- ..Hence, it is a counter-example to what we were trying to prove
- Typically indicates a bug found
- Models are translated back to Isabelle/HOL
 - Provides very valuable feedback!
- Not every counter-example is valid, however

Two kinds of bogus counter-examples

- ① Due to “Potential models”
 - Caused by:
 - Quantifiers
 - λ -expressions
 - These constructs render Yices’s logic incomplete
 - Clearly marked by Yices and the translator

Two kinds of bogus counter-examples

- 1 Due to “Potential models”
 - Caused by:
 - Quantifiers
 - λ -expressions
 - These constructs render Yices’s logic incomplete
 - Clearly marked by Yices and the translator
- 2 Due to uninterpreted terms and types
 - Caused by:
 - Lack of “auxiliary” lemmata
 - Lack of definitions of the functions used
 - These are more problematic..

- 1 Introduction
- 2 Connecting Isabelle to Yices
- 3 Example Translations**
- 4 Dealing with false alarms
- 5 Application: Verifying C programs
- 6 Summary

Reflexivity

```
lemma "x = x"  
by ismt
```

Reflexivity

```
lemma "x = x"  
by ismt
```

Generates

```
(define-type 'a)  
(define x::'a)  
(assert (/= x x))
```

- Monomorphisation in action!

No odd number is a multiple of 2

```
lemma "a = (2::int) * n + 1  $\longrightarrow$  a  $\neq$  2 * m"
```

```
by ismt
```

No odd number is a multiple of 2

```
lemma "a = (2::int) * n + 1  $\longrightarrow$  a  $\neq$  2 * m"  
by ismt
```

Generates

```
(define a::int)  
(define n::int)  
(define m::int)  
(assert (not (=> (= a (+ (* 2 n) 1))  
                 (/= a (* 2 m)))))
```

Counter examples

Absolute values

```
lemma "abs (n::int) = n"  
by ismt
```

Counter examples

Absolute values

```
lemma "abs (n::int) = n"  
by ismt
```

Generates

```
(define n::int)  
(assert (/= (if (< n 0)  
                (- 0 n) n)  
          n))
```

Counter examples

Absolute values

```
lemma "abs (n::int) = n"  
by ismt
```

Generates

```
(define n::int)  
(assert (/= (if (< n 0)  
              (- 0 n) n)  
         n))
```

Counter example

A counter-example is found:
n = -1

Quantification and Higher order functions

- Quantifiers can render Yices incomplete
- Not a problem if in universal prenex form

Quantification and Higher order functions

- Quantifiers can render Yices incomplete
- Not a problem if in universal prenex form

A trivial lemma

```
lemma " $\forall i f g. (f = g \longrightarrow f i = g i)$ "
```

Quantification and Higher order functions

- Quantifiers can render Yices incomplete
- Not a problem if in universal prenex form

A trivial lemma

```
lemma "∀i f g. (f = g → f i = g i)"
```

Generates

```
(define-type 'a)
(define-type 'b)
(define i::'a)
(define f::(-> 'a 'b))
(define g::(-> 'a 'b))
(assert (not (=> (= f g) (= (f i) (g i)))))
```

- automatically proven by Yices..

Quantification and Higher order functions (cont'd)

Counter-examples

```
lemma " $\forall i f g. (f i = g i \longrightarrow f = g)$ "
```

Quantification and Higher order functions (cont'd)

Counter-examples

```
lemma "∀i f g. (f i = g i → f = g)"
```

Generates

```
(define-type 'a)  
(define-type 'b)  
(define i::'a)  
(define f::(-> 'a 'b))  
(define g::(-> 'a 'b))  
(assert (not (=> (= (f i) (g i)) (= f g))))
```

Quantification and Higher order functions (cont'd)

Counter-examples

```
lemma "∀i f g. (f i = g i → f = g)"
```

Generates

```
(define-type 'a)
(define-type 'b)
(define i::'a)
(define f::(-> 'a 'b))
(define g::(-> 'a 'b))
(assert (not (=> (= (f i) (g i)) (= f g))))
```

Not true!

A counter-example is found:

`i = 1`

`f 1 = g 1`

Quantification and Higher order functions (cont'd)

Counter-examples

```
lemma "∀i f g. (f i = g i → f = g)"
```

Generates

```
(define-type 'a)
(define-type 'b)
(define i::'a)
(define f::(-> 'a 'b))
(define g::(-> 'a 'b))
(assert (not (=> (= (f i) (g i)) (= f g))))
```

Not true!

A counter-example is found:

```
i = ismt_const 1
```

```
f (ismt_const 1) = g (ismt_const 1)
```

Parameterized datatypes

Monomorphise as we go

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

Parameterized datatypes

Monomorphise as we go

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

```
lemma "Left False  $\neq$  Right (4::int)  
       $\wedge$  Left (1::nat)  $\neq$  Right x"
```

Monomorphise as we go

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

```
lemma "Left False  $\neq$  Right (4::int)  
       $\wedge$  Left (1::nat)  $\neq$  Right x"
```

- Types involved:
 - (bool \times int) Either
 - (nat \times 'a) Either

Parameterized datatypes (cont'd)

Polymorphic Either

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

Parameterized datatypes (cont'd)

Polymorphic Either

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

(bool × int) and (nat × 'a) instances

```
(define-type Either-bool-int  
  (datatype (Left-bool-int bool)  
            (Right-bool-int int)))
```

Parameterized datatypes (cont'd)

Polymorphic Either

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

(bool × int) and (nat × 'a) instances

```
(define-type Either-bool-int
  (datatype (Left-bool-int bool)
            (Right-bool-int int)))

(define-type 'a)

(define-type Either-nat-'a
  (datatype (Left-nat-'a nat)
            (Right-nat-'a 'a)))
```

Parameterized datatypes (cont'd)

Polymorphic Either

```
datatype ('a, 'b) Either = Left 'a | Right 'b
```

(bool × int) and (nat × 'a) instances

```
(define-type Either-bool-int
  (datatype (Left-bool-int bool)
            (Right-bool-int int)))

(define-type 'a)

(define-type Either-nat-'a
  (datatype (Left-nat-'a nat)
            (Right-nat-'a 'a)))
```

[automatically generated accessor functions not shown for clarity...]

What's supported?

- Basic strategy:
 - Translate to native Yices format whenever there is an *obvious* corresponding construct.
 - Otherwise, uninterpret.
- Supported types
 - `int`, `nat`, `bool`
 - `'a list`, `'a option`
 - Tuples
 - Records with polymorphic fields
 - Excluding extensible records
 - User defined datatypes, both parameterized and recursive
 - No mutual recursion
 - Functions: Both first-order and higher-order

Supported constants

- Equality: =
- Booleans: True, False, \leq , $<$, \longrightarrow , \implies , \vee , \wedge , \neg , and dvd.
- Arithmetic: +, -, \times , /, - (unary minus), div, mod, abs, Suc, min, max, fst, and snd.
- Arithmetic is understood both for nat and int
 - All other number types remain uninterpreted

Supported expressions and binding constructs

- If-expressions, let bindings, λ -abstractions,
- Quantifiers (\forall , \exists , \wedge),
- Case expressions over
 - Tuples
 - Naturals
 - Internal option type and lists
 - Arbitrary user defined types
- Function and record update expressions

What's *not* supported?

- HOL constructs
 - $\exists!$, Ball, Bex
 - Hilbert's choice (ϵ) and Least
 - Mutual recursion in datatypes
 - Extensible records
 - There are just no good Yices equivalents

What's *not* supported?

- HOL constructs
 - $\exists!$, Ball, Bex
 - Hilbert's choice (ϵ) and Least
 - Mutual recursion in datatypes
 - Extensible records
 - There are just no good Yices equivalents
- Types:
 - Fixed size bit-vectors and Rationals
 - We plan to add these as needed

What's *not* supported? (cont'd)

- Most importantly
 - No function definitions
 - No lemmas
- User's need to insert these manually
- Appropriate instances need to be chosen

What's *not* supported? (cont'd)

- Most importantly
 - No function definitions
 - No lemmas
- User's need to insert these manually
- Appropriate instances need to be chosen
- This is the major source of false alarms

- 1 Introduction
- 2 Connecting Isabelle to Yices
- 3 Example Translations
- 4 Dealing with false alarms**
- 5 Application: Verifying C programs
- 6 Summary

Computing the length of boolean-lists

```
consts len :: "bool list  $\Rightarrow$  nat"  
primrec "len [] = 0"  
        "len (x#xs) = 1 + len xs"
```

Uninterpreted functions

Computing the length of boolean-lists

```
consts len :: "bool list  $\Rightarrow$  nat"  
primrec "len [] = 0"  
        "len (x#xs) = 1 + len xs"
```

A trivial lemma

```
lemma "len [True, False] = 2"  
by ismt
```

The bogus counter-example

Response from ismt

A counter-example is found:

```
len [True, False] = 3
```

The bogus counter-example

Response from ismt

A counter-example is found:

```
len [True, False] = 3
```

The translation

```
(define-type list-bool
  (datatype Nil-bool
            (Cons-bool hd::bool tl::list-bool)))
```

The bogus counter-example

Response from ismt

A counter-example is found:

```
len [True, False] = 3
```

The translation

```
(define-type list-bool
  (datatype Nil-bool
            (Cons-bool hd::bool tl::list-bool)))
```

```
(define len::(-> list-bool nat))
```

The bogus counter-example

Response from ismt

A counter-example is found:

```
len [True, False] = 3
```

The translation

```
(define-type list-bool
  (datatype Nil-bool
            (Cons-bool hd::bool tl::list-bool)))

(define len::(-> list-bool nat))

(assert (/= (len (Cons-bool true
                 (Cons-bool false Nil-bool)))
           2))
```

Auxiliary Lemmata

```
lemma len0: "len [] = 0"
```

```
lemma len1: "len (x#xs) = 1 + len xs"
```

Auxiliary Lemmata

```
lemma len0: "len [] = 0"
```

```
lemma len1: "len (x#xs) = 1 + len xs"
```

insert before calling ismt

```
lemma "len [True, False] = 2"
```

```
apply (insert len0 len1)
```

```
by ismt
```

The current goal state

The top-most goal now looks like

$$\begin{aligned} & \llbracket \text{len } [] = 0; \bigwedge x \text{ xs. len } (x \# \text{xs}) = 1 + \text{len } \text{xs} \rrbracket \\ & \implies \text{len } [\text{True}, \text{False}] = 2 \end{aligned}$$

The current goal state

The top-most goal now looks like

$$\llbracket \text{len } [] = 0; \bigwedge x \text{ xs. len } (x \# \text{xs}) = 1 + \text{len } \text{xs} \rrbracket \\ \implies \text{len } [\text{True}, \text{False}] = 2$$

In addition, the tactic now generates

```
(assert (= (len Nil-bool) 0))
(assert
  (forall (x::bool)
    (forall (xs::list-bool)
      (= (len (Cons-bool x xs))
        (+ 1 (len xs))))))
```

The current goal state

The top-most goal now looks like

$$\llbracket \text{len } [] = 0; \bigwedge x \text{ xs. len } (x \# \text{xs}) = 1 + \text{len } \text{xs} \rrbracket \\ \implies \text{len } [\text{True}, \text{False}] = 2$$

In addition, the tactic now generates

```
(assert (= (len Nil-bool) 0))
(assert
  (forall (x::bool)
    (forall (xs::list-bool)
      (= (len (Cons-bool x xs))
        (+ 1 (len xs))))))
```

- The proof is now automatic!

- 1 Introduction
- 2 Connecting Isabelle to Yices
- 3 Example Translations
- 4 Dealing with false alarms
- 5 Application: Verifying C programs**
- 6 Summary

Buffer copy (CIL-like)

```
int dst[buf_size];
int *s; int *d;
s = src; d = dst;
while(1)
  if(*s == 0) break;
  else {
    *d = *s;
    s++;
    d++;
    continue;
  }
*d = 0;
```

Buffer copy (CIL-like)

```
int dst[buf_size];
int *s; int *d;
s = src; d = dst;
while(1)
  if(*s == 0) break;
  else {
    *d = *s;
    s++;
    d++;
    continue;
  }
*d = 0;
```

SeqC equivalent

```
(doSeqC { with_array buf_size (λ(pdst :: int Ptr).
  with_var (λ(pps :: int Ptr Ptr).
    with_var (λ(ppd :: int Ptr Ptr). doSeqC {
      assign_ptr pps psrc;
      assign_ptr ppd pdst;
      loopAsrt
        (loopInv False psrc pdst pps ppd buf_size)
        (loopInv True psrc pdst pps ppd buf_size)
        (λ r s. False)
      (doSeqC {ps ← deref_ptr pps;
        ct ← deref_ptr ps;
        if (ct = 0)
          then break
        else doSeqC {pd ← deref_ptr ppd;
          assign_ptr pd ct;
          assign_ptr pps (ps +p 1);
          assign_ptr ppd (ppd +p 1);
          continue}});
        pd ← deref_ptr ppd;
        assign_ptr pd 0;
        c_return 0
      })))
}))"
```

An Isabelle/HOL model of C

- Tactics to generate/propagate VC's
- Strategy: solve VC's using ismt

A typical VCG (abstracted)

```
definition vcg :: "addr  $\Rightarrow$  addr  $\Rightarrow$  addr  $\Rightarrow$  int  $\Rightarrow$   
                (addr  $\Rightarrow$  byte)  $\Rightarrow$  bool" where  
"vcg src dst s_ptr n h  
  = (let s = h s_ptr;  
      d = dst - src + s;  
      h' = h(d := h s, s_ptr := h s_ptr + 1)  
  in ( src  $\leq$  s  $\wedge$  is_str s (src + n - s) h  
       $\wedge$   $\neg$  s_ptr mem (str_addrs s n h)  
       $\wedge$   $\neg$  d mem (str_addrs s n h)  
       $\wedge$  h s  $\neq$  0  
       $\longrightarrow$   $\neg$  s_ptr mem (str_addrs (s+1) n h')))"
```

Experience with discharging VCs

- Needed to add lemmas for parameterized verification
- Manual instantiations were necessary
- Finding required lemmas:
 - Manual backchaining process
 - Prove and add extra subgoals as hypotheses as needed
- Counter-examples were helpful when they were *small*
 - Abstract-counter examples would be nice
 - Consider the model: $x = 3 \wedge y = 3 \wedge P\ 3$
 - If we know $P\ 3$ is false, we still can't tell:
 - Did Yices choose $x = 3$ to make P false?
 - Or, did it choose $y = 3$ to falsify P ?
 - We'd like to get " $P\ x$ " as an abstract counter-example
- Completely ground models are not too helpful

A note on speed

Prove: All solutions of $x_{i+2} = |x_{i+1}| - x_i$ are periodic with period 9

```
lemma "[ x3 = |x2|-x1; x4 = |x3|-x2; x5 = |x4|-x3;  
        x6 = |x5|-x4; x7 = |x6|-x5; x8 = |x7|-x6;  
        x9 = |x8|-x7; x10 = |x9|-x8; x11 = |x10|-x9 ]"  
  => x1 = x10 & x2 = (x11::int)"
```

⁰Example due to John Harrison

A note on speed

Prove: All solutions of $x_{i+2} = |x_{i+1}| - x_i$ are periodic with period 9

```
lemma "[ x3 = |x2|-x1; x4 = |x3|-x2; x5 = |x4|-x3;  
        x6 = |x5|-x4; x7 = |x6|-x5; x8 = |x7|-x6;  
        x9 = |x8|-x7; x10 = |x9|-x8; x11 = |x10|-x9 ]"  
  => x1 = x10 & x2 = (x11::int)"
```

- Isabelle's presburger tactic: 3.5 minutes
- Isabelle's arith tactic: 2.25 minutes.
- ismt tactic via Yices: < 1 second.
- Yices is blazing fast!

⁰Example due to John Harrison

- 1 Introduction
- 2 Connecting Isabelle to Yices
- 3 Example Translations
- 4 Dealing with false alarms
- 5 Application: Verifying C programs
- 6 Summary**

Summary and Future Work

- A practical connection between Yices and Isabelle
 - Great for “simpler” but tedious goals
 - **Not** a sledge-hammer!
- Counter-examples translated back to HOL
- Extremely valuable even in Oracle mode
 - Full proofs can be given later
 - Speeds up development time immensely
 - Full dumps provided for inspection
- Future work
 - Use counter-example info to identify false alarms
 - Automatically add needed definitions/lemmas
 - Support for more HOL constructs and types
 - Especially bit-vectors
 - Incrementality (using the programmatic API)

Thank you!

- Download ismt from:
`www.galois.com/company/open_source/ismt`
- Tested to work with Isabelle 2008 and Yices 1.0.13
- Free with a permissive BSD-style license
- Patches and improvements most welcome!