

Using Yices as an automated solver in Isabelle/HOL

Levent Erkök

John Matthews

Galois, Inc.
12725 SW Millikan Way, Ste 290
Beaverton, OR 97005
{levent.erkok,matthews}@galois.com

ABSTRACT

We describe our integration of the Yices SMT solver into the Isabelle theorem prover. This integration allows users to take advantage of the powerful SMT solving techniques within the interactive theorem proving environment of Isabelle, considerably increasing the automation level for a significant subset of Isabelle/HOL.

1. INTRODUCTION

This paper describes the Isabelle `ismt` tactic,¹ developed by Galois to seamlessly integrate the Yices SMT solver within the interactive theorem proving environment of Isabelle, thus increasing the automation level considerably as well as providing counterexample information back to the user when Yices detects a formula is invalid; similar to PVS’s `yices` strategy [6].

The `ismt` tactic is freely available on the internet with a permissive BSD-style license [5].

1.1 Yices

Yices is a modern SMT solver that supports uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit vectors, λ -expressions, and quantifiers [7, 11, 15]. Yices’s input language is based on a LISP like syntax extended with type declarations. Nevertheless, Yices’s input language is still significantly more restrictive than Isabelle/HOL [21]. For instance, Yices currently does not support parameterized datatype declarations, mutual or nested recursion in datatypes, or bounded quantification over sets. Most importantly, Isabelle/HOL’s and Yices’s type systems are substantially different: While the former has a polymorphic type system, Yices only allows monomorphic definitions with uninterpreted types. However, we still consider Yices a suitable target for integration,

¹The name `ismt` was chosen to avoid conflict with an already existing `smt` tactic (see Section 2.2), and also to emphasize our future plans of taking advantage of incrementality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AFM08: Automated Formal Methods '08 Princeton, New Jersey, USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

and our tactic is able to translate many of Isabelle/HOL’s extra features into Yices equivalents.

Most SMT solvers support the *SMT-Lib* language [9], which is the input format used in the SMT competitions. However, we chose to avoid translation through this medium for two reasons. First, SMT-Lib only supports a limited set of theories [8] which does not include datatypes, tuples, records, etc. Second, the SMT-Lib language is not incremental. That is, facts cannot be asserted or retracted in the middle of a proof. While we do not currently make any significant use of Yices’ native incremental API, our future plans do include taking full advantage of this functionality.

1.2 Modes of integration

The `ismt` tactic invokes Yices as an *external oracle*, meaning that it trusts the soundness of Yices and our translator. Whenever Isabelle produces a theorem via an external oracle, it attaches a *trust tag* to it, as well as to any other theorem that uses this theorem in its proof. The trust tag says which tool was invoked, and the formula the tool proved. Isabelle displays a “[!]” annotation on any theorem containing trust tags, and the list of trust tags associated with a theorem can also be directly queried.

Assuming Yices supports a mode in the future where explicit proof objects are returned, then we would also like to build a *proof replay* mode for `ismt` where the proof object is used to reconstruct a purely Isabelle proof of the theorem that would not contain any trust tags.

2. RELATED WORK

There have been several attempts at integrating SMT solvers into theorem proving environments. In this section we review the most relevant ones to our work.

2.1 Integration of Yices with PVS

The Yices SMT solver can be used as an end-game solver in PVS [6]. The translation from PVS to Yices is much more direct than ours, since Yices and PVS share the same type system. Similar to our work, Yices acts as a trusted solver in PVS, whose results are not verified independently. Unlike our tactic, however, the models generated by Yices are not translated back to PVS notation: A failed proof attempt by Yices is simply interpreted by PVS as a skip, having no effect on the proof state.

2.2 The `smt` tactic

Barsotti et al. describes how to integrate generic SMT solvers with Isabelle [10, 14]. Similar to our work, their tac-

tic works as an oracle as well, i.e., no proof reconstruction is done. Unlike our work, however, they target the SMT-Lib language [9] as their translation medium, allowing them to use arbitrary SMT solvers that support this common language. Due to the limitations of SMT-Lib, however, Bartsotti et al. forgo expressive power, not being able to support datatypes, case-statements, λ -expressions, tuples, records, etc., which are the basic pillars of the Isabelle/HOL language. We consider the expressive power afforded by the richer internal language of individual SMT solvers well worth the cost of building custom translators.

2.3 The rv tactic

Fontaine et al. describe how to perform proof reconstruction using proof generating SMT solvers [17]. In their work, they use haRVey [3] to generate “proof hints,” that can later be replayed by Isabelle via the `rv` tactic. While this is precisely the technique we would like to use in the future, currently haRVey only supports propositional logic, uninterpreted functions, and linear arithmetic; a rather limited language compared to what our tactic can handle.

2.4 Integration of CVC-Lite with HOL-Light

The final related work we would like to review is the integration of CVC-lite SMT solver [1] within HOL-Light [4], as described by McLaughlin, Barrett, and Ge [20]. In this work, proofs generated by CVC-Lite are represented as tree-like data structures [13], which are parsed back and replayed in the HOL-Light theorem prover. This translation of proof trees is especially made easy since the CVC-Lite logic is very close to a subset of the HOL-Light logic.

McLaughlin et al. point out that to integrate the CVC-Lite proofs about arrays, for instance, they had to extend HOL-Light to understand CVC-Lite inference rules. While they note that this was a trivial task in this particular case, it is not clear how easy it would be to repeat this exercise to other theories of interest.²

3. HOW THE ISMT TACTIC WORKS

The `ismt` tactic proves theorems by having Yices prove their negations unsatisfiable. When invoked, the `ismt` tactic performs the following tasks on the topmost subgoal of the Isabelle goal stack:

- Translate the types occurring in the formula into Yices type declarations. This process requires monomorphisation of HOL datatypes, records, etc., to map polymorphic operators and type variables to corresponding monomorphic versions supported by Yices. (See Section 4 for details.)
- Negate the subgoal and translate it to Yices. If a HOL constant has no corresponding Yices construct, then declare it as an uninterpreted constant of the appropriate type. For instance, let `isEven :: nat \Rightarrow bool` be a user defined HOL constant. Then, for the HOL expression `isEven (4::nat)`, we would generate the following Yices translation:

```
(define isEven::(-> nat bool))
(assert (not (isEven 4)))
```

where no defining equations for `isEven` are added (however the user can always first insert the defining equations into the original subgoal as quantified lemmas).

- Pass the generated script to Yices. If Yices returns a model (i.e., a set of assignments that satisfies the negation of the input), we turn that into a refutation of the original formula. Naturally, the Isabelle proof attempt fails at this point. (This counterexample might be spurious, due to the presence of uninterpreted constants. We will discuss this possibility in detail in Section 5.)
- If Yices determines the clauses are unsatisfiable, then trigger Isabelle’s oracle mechanism and accept the original subgoal formula as a (trust-tagged) theorem.

The following HOL entities are properly understood and translated by the `ismt` tactic to Yices’s internal language. Any other construct will be translated as an uninterpreted constant or type:

- **Types.** Ground types: `int`, `nat`, `bool`. Basic HOL types: polymorphic lists, option type, tuples of arbitrary arity, including `unit`. Records with polymorphic fields (except extensible records). User defined datatype declarations: Both parameterized and recursive variants are supported. (However, they cannot be mutually recursive, either directly or indirectly via nesting.) Functions: Both first-order and higher-order functions are supported.
- **Constants.** Equality: `=`, supported polymorphically at all types. Boolean operators: `True`, `False`, `\leq` , `$<$` , `\rightarrow` , `\implies` , `\vee` , `\wedge` , `\neg` , and `dvd`. Operators:³ `+`, `-`, `\times` , `/`, `-` (unary minus), `div`, `mod`, `abs`, `Suc`, `min`, `max`, `fst`, and `snd`.
- **Expressions and binding constructs:** If-expressions, let bindings, λ -abstractions, quantifiers (`\forall` , `\exists` , `\bigwedge`), case expressions (over tuples, naturals, option type, lists, and arbitrary user defined types), function and record update expressions.

4. EXAMPLES

In this section we will walk over a number of example uses of the `ismt` tactic, demonstrating its basic capabilities.

4.1 Basics

Consider the classic excluded-middle example:

```
lemma "a  $\vee$   $\neg$ a"
by ismt
```

Running Isabelle on this input yields:

³The arithmetic operators (`+`, `-`, etc.), and comparisons (`\leq` , `$<$`) are supported both at their `int` and `nat` instances. Use of arithmetic operators at other Isabelle numeric types will remain uninterpreted. Also note that Yices only supports linear-arithmetic i.e., multiplication/division can only be done by a constant. If a non-linear expression is given to the translator, it will still be translated, but Yices will reject the input with a failure message.

² An e-mail inquiry to CVC developers revealed that while their latest SMT solver (named CVC3) has support for proof-generation facilities, the documentation remains sparse [2]. Also of question is the continued support for this feature as CVC itself evolves.

```
lemma ?a ∨ ¬?a [!]
```

Note that Isabelle tags the free variable `a` with `?`, indicating it is a *schematic variable*, i.e., the lemma is proven for all possible substitutions of the variable `a`. Furthermore, the lemma is displayed with a `[!]` annotation, indicating the role of the oracle.

Here's a slightly more interesting example, showing that an odd number can not be a multiple of 2:

```
lemma "a = (2::int) * n + 1 ⟶ a ≠ 2 * m"
by ismt
```

which will be proven by the `ismt` tactic directly. Note that the type qualifier on 2, (i.e., `2::int`) is necessary. Otherwise Isabelle's overloaded numbers would have caused this statement to have a more polymorphic type than intended, and the translator would have left the arithmetic operators uninterpreted.

If the input to the `ismt` tactic is not a Yices-theorem, then a counterexample will be generated. Consider:⁴

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

Running Isabelle on this input will yield:

```
*** A counter-example is found:
***      n = -1
```

Counterexample generation raises an interesting question in the presence of uninterpreted constants. Consider the following example, where we do not indicate what specific type the expression `None` has.

```
lemma "n = None"
by (ismt model: abort)
```

The tactic will respond with:

```
*** A counter-example is found:
***      Some (ismt_const 1) = n
```

The counterexample uses the function `ismt_const`. The following excerpt from the generated Yices code might help explain the need for this constant:

```
(define-type 'a)
(define-type option-'a
  (datatype None-'a (Some-'a the::'a)))
(define n::option-'a)
(assert (/= n None-'a))
```

The type assigned to the HOL constant `n` is `'a option`, which results in a rendering of the `option` type at the uninterpreted type `'a`. (See Section 4.5 for details on how datatype declarations are translated.) Yices assumes all uninterpreted types are integers when generating models. When the HOL counterexample is generated from the Yices model, however, we cannot use these integers as the values of the corresponding variables; doing so would not be type correct. Hence, we have defined an uninterpreted HOL constant `ismt_const` with the type `int ⇒ 'a`. Such counterexamples should be read such that the arguments to `ismt_const` are indices into the right HOL type, where different indices pick different values in the corresponding domain. (That is, `ismt_const` should be considered an injective function.)

⁴The `model: abort` flag instructs `ismt` to throw an exception when a model is returned by Yices. The other possible options are `silent`, which acts as skip; and the default `notify`, which is the same as skip except it also displays the counterexample in Isabelle's trace buffer.

4.2 Anonymous functions

HOL's λ -abstractions are compiled into their Yices counterparts. Here are several examples:⁵

```
lemma "(λx::int. x+2) = (λy. 2+y)"
lemma "(λx::int. x+2) = f"
```

The first lemma generates the following Yices code:

```
(assert (/= (lambda (x::int) (+ x 2))
            (lambda (y::int) (+ 2 y))))
```

which is automatically proven.

The second lemma generates the following code:

```
(define f::(-> int int))
(assert (/= (lambda (x::int) (+ x 2)) f))
```

which causes the `ismt` tactic to generate the following counterexample:

```
*** A counter-example is found:
***      f -1 = 2
```

Note that the use of λ -expressions can trigger incompleteness, though we have found this to occur rarely in practice. (See Section 6 for details).

4.3 Tuples

HOL tuples are converted to their Yices counterparts appropriately. Tuples of arbitrary arity are implemented by right-nested tuples in HOL, and the translator uses the same technique to represent them in Yices.

The following lemma is automatically proven:

```
lemma "(x, y, x) = (y, x, y) ⟶ x = y"
```

It generates the code:

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

Projections `fst` and `snd` are translated accordingly:

```
lemma "fst t = snd t ⟶ (snd t, fst t) = t"
```

This lemma generates the following code:

```
(define-type 'a)
(define t::(tuple 'a 'a))
(assert (= (select t 1) (select t 2)))
(assert (/= (mk-tuple (select t 2) (select t 1))
          t))
```

and is automatically proven by Yices.

Any models produced by Yices for the negation of the goal are automatically translated back to HOL. Consider:

```
lemma "snd (f, f True) = False ⟶ f False = True"
```

which generates:

⁵For brevity, we will no longer show the actual call to `ismt`, i.e., each lemma line should be followed by the command `by (ismt model: abort)`.

```
(define f::(-> bool bool))
(assert (= (select (mk-tuple f (f true)) 2) false))
(assert (/= (f false) true))
```

We get the following HOL counterexample:

```
*** A counter-example is found:
***   f True = False
***   f False = False
```

Notice that the components of a tuple can contain arbitrary elements, including functions.

4.4 Let expressions

HOL let-expressions are implemented by the higher order function `Let :: 'a ⇒ ('a ⇒ 'b) ⇒ 'b`, so that `let x = 1 in x + x` is syntactic sugar for `Let 1 (λx. x + x)`. We convert these directly to Yices let-expressions. Here is an example:

```
lemma "let x = (1::int) in let y = 2 in x+y = 3"
```

Our tactic generates the following Yices code:

```
(assert (not (let ((x::int 1))
                 (let ((y::int 2))
                     (= (+ x y) 3))))))
```

which is automatically proven.

Bound variables in a let-expression can be of arbitrary types; the translator will ensure appropriate eta-expansion is done to preserve Yices' stringent function arity requirements, as demonstrated below:

```
lemma "(3::int) = (let f = (op +) 2 in f 1)"
```

This lemma will be automatically proven by `ismt`. It generates the following code:

```
(assert (/= 3 (let ((f::(-> int int))
                  (lambda (etav::int)
                    (+ 2 etav))))
         (f 1))))
```

Note the eta-expansion in the definition of `f` to make sure that the Yices constant `+` is applied to the correct number of arguments.

4.5 Datatype declarations

One of the ubiquitous aspects of functional programming (either in HOL or in any other functional language) is the use of datatype declarations. The `ismt` tactic translates (most) datatype declarations to their Yices counterparts. There are several obstacles, however:

- Yices does not support parameterized or polymorphic datatype declarations. Our translator “flattens-out” the use of parameterized datatypes on the fly, generating individual monomorphised instances.
- Yices does not allow datatype declarations to be mutually recursive, either directly (via the use of two datatype declarations), or indirectly (via the use of nested recursion). Such cases are detected by the translator and rejected.

4.5.1 Datatypes without parameters

Any non-parameterized datatype, recursive or otherwise, generates an equivalent declaration in Yices.

Simple enumerations are converted to `scalar` declarations:

```
datatype Kind = Odd | Even
```

gets translated to:

```
(define-type Kind (scalar Odd Even))
```

Recursive types are translated accordingly:

```
datatype Nat = Zero | Succ Nat
```

gets translated to:

```
(define-type Nat (datatype Zero (Succ aSucc::Nat)))
```

Unlike HOL, Yices requires all fields in a `datatype` declaration to have associated accessors. The accessor `aSucc :: Nat ⇒ Nat` was automatically generated in the last example to satisfy this requirement. (The translator also allows users to register their own custom accessors, see [16] for details.)

4.5.2 Parameterized datatypes

Each use of a parameterized datatype at a different instance causes the translator to generate a new set of declarations. We call this translation the process of *monomorphisation*. To illustrate, consider the following lemma:

```
datatype ('a,'b) Either = Left 'a | Right 'b
lemma "Left False ≠ Right (4::int)
      ∧ Left True ≠ Right x"
```

causes the translator to generate the following code:

```
(define-type 'a)
(define-type
  Either-bool-int
  (datatype
    (Left-bool-int aLeft-bool-int::bool)
    (Right-bool-int aRight-bool-int::int)))
(define-type
  Either-bool-'a
  (datatype (Left-bool-'a aLeft-bool-'a::bool)
            (Right-bool-'a aRight-bool-'a::'a)))
(define x::'a)
(assert (not (and (/= (Left-bool-int false)
                     (Right-bool-int 4))
                 (/= (Left-bool-'a true)
                     (Right-bool-'a x))))))
```

which is successfully proven by Yices. Note that each distinct use of the `Either` type caused a new datatype declaration, including the case where there is a free type variable. The names of types are used with dashes to create unique constructor names, as in `Left-bool-int` or `Right-bool-'a`.

Recursive declarations are translated similarly:

```
datatype 'a Tree = Leaf 'a
                  | Branch "'a Tree" "'a Tree"
lemma "Leaf 3 ≠ Leaf (2::int)"
```

causes the translator to generate the following:

```
(define-type
  Tree-int
  (datatype (Leaf-int aLeaf-int::int)
            (Branch-int aBranch-int1::Tree-int
                       aBranch-int2::Tree-int)))
(assert (= (Leaf-int 3) (Leaf-int 2)))
```

which is again proven unsatisfiable automatically by Yices.

4.5.3 Direct and nested mutual recursion

Mutually recursive datatypes are not supported by the translator, due to the fact that Yices has no support for such constructs and there is no simple translation that can be applied in such cases. An example of such a declaration is the following recursive pair of datatypes:

```
datatype 'a AExp = Var 'a | BExp "'a AExp"
and      'a BExp = And "'a AExp" "'a AExp"
```

If a lemma involving one of the types `AExp` or `BExp` is sent to `ismt`, it will throw an exception rejecting the mutually recursive datatype declaration.

Nested recursion is another source for the same problem:

```
datatype 'a Term = Variable 'a | App "'a Term list"
```

In this case, the recursion for the `Term` datatype happens at the type `'a Term list` instead of the required type `'a Term`. Similar to the above case, nested recursive declarations will be rejected by `ismt` as well.

4.6 Case expressions

In HOL, every datatype declaration is accompanied by a corresponding case-construct to take the constructed terms apart. Unfortunately, Yices does not support case expressions natively. However, Yices does provide recognizers for each constructor in a datatype declaration, and we take advantage of this facility to compile down HOL case-expressions to a cascaded sequence of if-then-else expressions. Using this technique, we not only support case expressions over built-in types such as tuples, booleans, and lists, but also user-defined datatypes as well. To illustrate, consider the following lemma:

```
lemma "fst tp = (case tp of (x, y) => x)"
```

which can automatically be proven by `ismt`. It generates the following code:

```
(define-type 'a)
(define-type 'b)
(define tp::(tuple 'a 'b))
(assert (/= (select tp 1)
            ((lambda (x::'a) (lambda (y::'b) x))
             (select tp 1)) (select tp 2))))
```

Note that there is no explicit if-statement used in this case, since there is exactly one way to take a tuple apart.

Case expressions over naturals demonstrates the use of if-expressions:

```
lemma "6 = (case (5::nat) of 0 => 1 | Suc m => m+2)"
```

Yices can automatically prove the generated assertion for this case:

```
(assert (/= 6 (if (= 5 0)
                  1
                  ((lambda (m::nat) (+ m 2)) 4))))
```

HOL lists are slightly more interesting. Consider:

```
lemma "case [True, False] of
  [] => True
  | (y#ys) => y"
```

This lemma is automatically proven, generating the code:

```
(define-type
  list-bool
  (datatype Nil-bool (Cons-bool hd::bool
                               tl::list-bool)))
(assert
  (not
   (let
     ((case::list-bool
       (Cons-bool true (Cons-bool false
                       Nil-bool))))
     (if (Nil-bool? case) true
         ((lambda (y::bool)
            (lambda (ys::list-bool) y))
          (hd case))
         (tl case))))))
```

The translator uses a let-expression to wrap the test expression around (unless it is already a variable), as demonstrated above using the variable `casev`. This aids greatly in readability as it avoids duplicating the expression later on.

4.7 Records

HOL record types are translated into Yices records. HOL's extensible records are not supported, however, since there is no corresponding Yices construct.⁶

Consider the HOL lemma:

```
record pt = pt_x :: int
lemma "pt_x (| pt_x = 3 |) = 3"
```

We generate the following code:

```
(define-type pt (record pt_x::int))
(define-type unit (scalar Unity))
(assert (/= (select (mk-record pt_x::3) pt_x) 3))
```

(Note the appearance of the `unit` type in the output, which seems spurious. It, in fact, corresponds to the `more` field of the HOL record.)

Parameterized and polymorphic fields are converted as usual, by monomorphising them appropriately:

```
record ('a, 'b) pt2 =
  pt2_x :: 'a
  pt2_y :: 'b
  pt2_z :: int
```

```
lemma "pt2_x (| pt2_x = v, pt2_y = q,
               pt2_z = s |) = v"
```

The generated Yices code is:

```
(define-type 'a)
(define-type 'b)
(define-type pt2-'a-'b
  (record pt2_x::'a pt2_y::'b pt2_z::int))
(define-type unit (scalar Unity))
(define v::'a)
(define q::'b)
(define s::int)
(assert (/= (select (mk-record pt2_x::v
                        pt2_y::q pt2_z::s)
                  pt2_x)
            v))
```

⁶We plan to overcome this limitation in a future version by “flattening,” i.e., either by compiling HOL's extensible records to Yices records that contain all the relevant fields, or by compiling them into nested records.

HOL and Yices records are both extensional, allowing us to prove record equality theorems. Consider the following lemma that uses the `pt2` record as defined above:

```
lemma "(| pt2_x = a, pt2_y = b, pt2_z = a+b |)
      = (| pt2_x = b, pt2_y = a, pt2_z = c |)
      ⇒ a = b & c - b = a"
```

It generates the following additional code:

```
(define a::int)
(define b::int)
(define c::int)
(assert
  (= (mk-record pt2_x::a pt2_y::b pt2_z::(+ a b))
     (mk-record pt2_x::b pt2_y::a pt2_z::c)))
(assert (not (and (= a b) (= (- c b) a))))
```

which is proven automatically by Yices.

Finally, counterexamples will be translated back to their HOL counterparts:

```
record fr =
  f :: "int => int"
```

```
lemma "r1 = (r2 :: fr)"
```

A call to `ismt` generates the following counterexample:

```
*** A counter-example is found:
***   (| f = f r1 |) = r1
***   (| f = f r2 |) = r2
***   f r1 1 = 2
***   f r2 1 = 3
```

4.8 Function and record updates

HOL's function and record update notations are fully supported by the translator. Consider the following trivial function update theorem:

```
lemma "(f(i:=n)) i = n"
```

This lemma is successfully proven. It generates the following code:

```
(define-type 'a)
(define-type 'b)
(define f::(-> 'b 'a))
(define i::'b)
(define n::'a)
(assert (/= ((update f (i) n) i) n))
```

Record updates are similarly translated to their Yices equivalents.

4.9 Quantifiers

Quantifiers are a soft spot for SMT solvers, as they typically render the underlying algorithms incomplete. (We will return back to this point in Section 6 in detail.)

Our `ismt` tactic translates both meta- and object-level Isabelle quantifiers into Yices' input format. One optimization is that top-level universally bound variables are skolemized into top-level Yices uninterpreted constants. To illustrate, consider the following trivial lemma:

```
lemma "∧ x. (∀y. (x = x ∧ y = y))"
```

When `ismt` is invoked, it generates the following code that contains no quantifiers at all:

```
(define x::'a)
(define y::'b)
(assert (not (and (= x x) (= y y))))
```

Needless to say, Yices deduces unsatisfiability instantly.

When quantifiers are nested, however, the translator can no longer compile them away. In such cases, we simply translate them to their Yices equivalents. Consider the lemma:

```
lemma "∧x. [ (∀y. p y ⇒ q y) ] ⇒ p x → q x"
```

This lemma is proven successfully by Yices. The generated code is:

```
(define-type 'a)
(define x::'a)
(define p::(-> 'a bool))
(define q::(-> 'a bool))
(assert (forall (y::'a) (⇒ (p y) (q y))))
(assert (not (⇒ (p x) (q x))))
```

Note, in particular, how the parameter `x` becomes a top-level definition, while `y` remains `forall` bound in the Yices translation.

The treatment of the \exists binder is similar, except that top-level occurrences cannot be compiled away to top-level definitions. Here is a simple example to illustrate:

```
lemma "∃x. x > (0::nat)"
```

This lemma is proven automatically by Yices. It generates the code:

```
(assert (not (exists (x::nat) (< 0 x))))
```

The translator does not support the unique-existence quantifier ($\exists!$). The bounded versions of the quantifiers (`Ball` and `Bex`) are not supported either, and neither are the Hilbert's choice (ϵ), and the `Least` binders. Uses of these constructs will remain uninterpreted during the translation process. (It might be possible to support bounded quantifiers through Yices' predicate-subtyping. However, we currently refrain from this since Yices does not ensure type-correctness when predicate-subtyping is used. In particular, it is possible to define empty types in Yices, and exploit these to prove bogus theorems.)

5. DEALING WITH FALSE ALARMS

Due to the fact that certain constants will remain uninterpreted during the translation, the `ismt` tactic can come up with bogus counterexamples. In this section we consider two particular instances of this problem and discuss mitigations.

5.1 Recursive uninterpreted constants

Although non-recursive uninterpreted functions can be dealt with by unfolding their definitions before calling `ismt`, a different approach is needed when the functions are recursively defined. To illustrate, consider the function `len` below, which computes the length of boolean lists:

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

Consider the following lemma:

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

which generates the following code:

```
(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))
```

Yices provides the following counterexample:

```
*** A counter-example is found:
***   len [True, False] = 3
```

which is clearly bogus. The problem arises since we have not told Yices anything about the function `len`, leaving it uninterpreted.

There are clearly easier ways to prove this lemma in Isabelle, (in fact, a simply application of `auto` would suffice), but our goal is to show how additional quantified hypotheses can be added so that `ismt` can prove the lemma successfully. In this case, all we need to do is to assert the pattern-matching rewrite rules for `len` as extra Isabelle lemmas:

```
lemma len0: "len [] = 0"
lemma len1: "len (x#xs) = 1 + len xs"
```

Both of these lemmas can be proven by Isabelle’s `auto` tactic. We can now use these additional facts to guide Yices:

```
lemma "len [True, False] = 2"
apply (insert len0 len1)
```

The goal state after the `insert` tactic looks like:

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

When we apply `ismt` at this proof state, the generated Yices code looks like:

```
(define-type
  list-bool
  (datatype Nil-bool (Cons-bool hd::bool
    tl::list-bool)))
(define len::(-> list-bool nat))
(assert (= (len Nil-bool) 0))
(assert
  (forall (x::bool)
    (forall (xs::list-bool)
      (= (len (Cons-bool x xs))
        (+ 1 (len xs))))))
(assert (/= (len (Cons-bool true
  (Cons-bool false Nil-bool)))
  2))
```

which is easily decided by Yices to be unsatisfiable, allowing us to conclude that the original formula is indeed a theorem.

Unfortunately, not all false alarms can be dealt with using these techniques. There will invariably be certain constructs that will go uninterpreted during the translation. (Consider, for instance, more complicated recursive definitions where finding such “helper lemmata” would amount to proving the original theorem. Or HOL constructs such as

Hilbert’s choice operator that has no corresponding “executable” counterpart that we can use in the simplification process.) While these techniques can be helpful, our experiences with the `ismt` tactic suggest that such cases are best dealt within the theorem proving framework of Isabelle, instead of relying on a backend SMT solver.

6. INCOMPLETENESS

SMT solvers typically support richer languages/logics than they can actually decide. For instance, it is well known that quantifiers (i.e., \forall , \exists), and λ -expressions make logics incomplete. In such cases, the underlying solver typically returns a satisfying model as well, but there is a chance that this model might be bogus. (Such problems are reported by Yices as “unknown” to indicate this possibility.)

To illustrate, consider:

```
lemma "(( $\lambda(x::\text{bool}). f x$ ) = ( $\lambda x. \text{True}$ ))  $\implies f x$ "
```

The generated file contains (excerpt shown below):

```
(assert (= (lambda (x::bool) (f x))
  (lambda (x::bool) true)))
(assert (not (f x)))
```

We get the following “potential” counterexample from `ismt`, which is actually bogus in this particular case:

```
Potential HOL counterexample:
  x = False
  f False = False
```

Notice that incompleteness will *never* cause `ismt` to prove a non-theorem. Rather, it might prevent it from proving a valid assertion. In other words, soundness is never at risk due to this limitation.

7. EXPERIMENTS

Having discussed the `ismt` tactic in detail, we will now briefly turn to our use cases for it at Galois, providing examples from various projects, both past and present.

Galois is building several *cross-domain* web service applications that provide strong security guarantees about the confidentiality of data across separate security domains. For example, our *Trusted Services Engine* (TSE) is a multi-level secure filestore providing a single common repository for files and directories, where each user’s view of the file system is restricted according to that user’s security level.

Most of the TSE is written in the Haskell functional programming language, which provides a number of security benefits. However, we implemented the most security-critical cross-domain portion of the TSE as an 800-line C program, to eliminate any dependency on Haskell’s runtime system. In order to attain the highest government security certifications for the TSE, we decided to formally model this component in Isabelle and verify its memory safety and information flow properties [18]. Although the verification was successful, discovering the required inductive safety invariants and view relations was very labor-intensive.

7.1 C program verification

To speed up proofs for future cross-domain C components we have prototyped a monadic-style sequential semantics for a larger subset of C, called `SeqC`. Although `SeqC`’s semantics

still only covers a small portion of the C standard, it includes general `while` loops, the non-local control flow statements `return`, `break`, and `continue`, access control permissions for each memory byte, the ability to take addresses of C globals, locals, and malloc'ed memory, nondeterministically-modeled C functions such as `malloc` and `free`, and general `assume` and `assert` statements. `SeqC` also contains a non-recoverable `Err` state that is transitioned into upon any memory-safety violation or assertion failure. Memory-safety is thus defined as unreachability of `Err`, with `assume` statements first pruning out all execution paths that don't satisfy a program's environmental assumptions.

We have proved Hoare logic rules for `SeqC`, built a simple verification condition generator (VCG) as an Isabelle tactic, and run some initial experiments verifying memory-safety of small example C programs. So far we have found the `ismt` tactic to be quite helpful not only in proving the verification conditions, but also for debugging too-weak preconditions and loop invariants by inspecting the counterexamples returned. To keep the counterexample sizes tractable, we initially put small concrete bounds on program parameters such as array sizes. We then used selected rewrite rules and a high-level Isabelle tactic to automatically expand away the (now bounded) quantifiers and recursive functions in the formula before calling `ismt`. Once we had found the appropriate rewrites, preconditions, and loop invariants, the verification of our example programs was completely automatic, thanks to the versatility of our `ismt` tactic and the power of Yices.

To discover the necessary rewrite rules, we first had to figure out which part of the verification condition `ismt` couldn't solve. We wrote an Isabelle proof script that used case-splitting tactics to eliminate any rigid quantifiers (i.e. quantifiers that didn't require any witnesses to be invented), as well as top-level conjunctions and disjunctions in the proof goal. The result was one or more smaller subgoals that were jointly equivalent to the original proof goal. Then, we used the `ismt` tactic on each subgoal until it found one that failed.

7.2 Parameterized program verification

We have also used the `ismt` tactic in verifying a string-copy routine where the source string can either be on the heap or the C stack, with a precondition that the string length is not larger than the destination string buffer, nor aliased to the destination buffer or any of the program's local variables.

When we tried to re-run the verification where the source and destination buffer sizes were not fixed beforehand, the resulting goals required Yices to reason about quantified assertions. Unfortunately, we have found that Yices' quantifier instantiation heuristics were not up to the task. Furthermore, Yices currently does not allow users to specify their own domain-specific quantifier instantiation term patterns.

To illustrate the issues we encountered, below we have defined `vcg`, a simplified version of one of the parameterized subgoals generated by our case-splitting tactic that Yices was not able to solve, even with the appropriate quantified lemmas. (The combined Isabelle and `ismt` proof of this formula is part of a self-contained example file `vcg.thy`, included in our `ismt` release [5].)

definition

```
vcg :: "addr ⇒ addr ⇒ addr ⇒ int ⇒
      (addr ⇒ byte) ⇒ 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 ≤ s ∧ is_str s (src + n - s) h
       ∧ ¬ s_ptr mem (str_addrs s n h)
       ∧ ¬ d mem (str_addrs s n h)
       ∧ h s ≠ 0
       ⇒ ¬ s_ptr mem (str_addrs (s+1) n h')))"
```

Parameter `src` points to the start of the source string buffer, `dst` points to a destination buffer of size `n`, and `s_ptr` is `&s`, the address of the C variable `s`, which itself points to the next byte to copy from the source string. The variable `h` is the contents of the memory heap at the top of the loop. In the formula we have defined `d` to point to the destination byte that `*s` will be copied to, and `h'` to be the updated heap at the bottom of the current loop iteration where `*s` has been copied to `*d` with `s` incremented.

The function `vcg` mentions three recursively-defined functions. (i) The predicate `is_str` has type `addr ⇒ int ⇒ (addr ⇒ byte) ⇒ bool`; `is_str p n h` is true whenever `p` points to a null-terminated string in heap `h` that is no more than `n` bytes long, including the null byte. (ii) The function `str_addrs` has type `addr ⇒ int ⇒ (addr ⇒ byte) ⇒ addr list`. The call `str_addrs a n h` returns a maximum length `n` list of contiguous addresses in `h` starting at `a` up to and including the first address pointing to a null byte (if any). It represents the set of addresses that can alias the string. And (iii), the infix binary relation `_ mem _ :: 'a ⇒ 'a list ⇒ bool` is list membership. The types `addr` and `byte` are synonyms for `int`.

To make this example easier for `ismt`, we manually removed from `vcg` any hypotheses that were irrelevant to the conclusion. The remaining hypotheses are: At the top of the while loop `s` points to a string that fits within the `n`-byte source buffer `src`, `&s` and `d` are not aliased to the string, and `*s` is not null. The conclusion to verify is that at the end of the while loop `&s` is still not aliased to the string now pointed to by `s`. (This could happen if the while loop overwrites the source string's null byte.)

Discovering quantifier instantiations. We needed to give `ismt` four quantified lemmas for it to verify the formula. However we had to manually instantiate the following lemma with the substitutions $\{p' \leftarrow d, x \leftarrow h\ s\}$ and $\{p' \leftarrow s_ptr, x \leftarrow s + 1\}$, to keep `ismt` from timing out. The lemma variables `p`, `n`, and `h` remained quantified. With these instantiations, `ismt` is able to verify `vcg` in less than a second.

lemma str_addrs_simp:

```
"¬(p' mem str_addrs p n h) ∨ ((h p'=0) = (x=0))
 ⇒ str_addrs p n (h(p' := x)) = str_addrs p n h"
```

To find the required lemmas and instantiations, we started a manual backchaining process, where we asserted additional formulas as new Isabelle subgoals that we believed to be true. We confirmed this by running `ismt` on `vcg`, with the additional formulas as extra hypotheses. We then recursively followed this debugging process on each new Isabelle subgoal, until it was clear what extra lemma or instantiation was needed. (The file `vcg.thy` included in the distribution contains the additional formulas we used, specifically in the proof of `detailed_vcg_lemma` [5].)

The counterexamples generated by `ismt` were helpful for debugging these subgoals if they were small enough. However, we often found it quicker to ignore the counterexample and instead inspect the abstract Isabelle subgoal. This was because Yices would typically assign multiple variables the same concrete value in the counterexample, e.g., $x = 3 \wedge y = 3$. If the counterexample also asserted a formula $(P \ 3)$ that we knew to be false, we couldn't then tell whether Yices had chosen to satisfy $(P \ x)$ or $(P \ y)$. But this information was usually necessary to determine which quantified lemma would eliminate the counterexample for all possible values of x or y in the subgoal. It would be very helpful to have a Yices command to return an “abstract” counterexample that displays the occurrences of free variables in formulas and subterms, rather than substituting in their concrete values.

8. TIPS FOR USING ISMT

The following list summarizes a number of tips reflecting our experiences with the `ismt` tactic.

- *Avoid nested quantifiers.* The translator will generate separate top-level `assert` statements for each quantified hypothesis found in the subgoal. This is preferable as it seems to enable more of Yices quantifier instantiation heuristics. Try to lift quantified formulas into top-level hypotheses whenever possible.
- *Restrict arithmetic to `nat`'s and `int`'s.* While Isabelle allows arithmetic over arbitrary types (using axiomatic type-classes), the Yices backend is not rich enough to understand such constructs. When such goals are sent to `ismt`, it is very likely that a bogus counterexample will be returned, since these number types will remain uninterpreted. Try to restrict arithmetic to `nat`'s and `int`'s only, which are fully supported.
- *Watch for “uninterpreted” constants.* Pay close attention to the counterexamples returned by `ismt`.⁷ It is likely that the falsifying model will be bogus due to unknown facts about these constants. If possible, apply the techniques described in Section 5 to resolve such cases.
- *Be type-specific, especially when using records.* Isabelle rewrite rules regarding records tend to be too polymorphic, applying to a variety of record types. Since the translator does not support extensible records (see Section 4.7), such rewrite rules can create subgoals that are unnecessarily more general than needed. Whenever possible, provide sufficient type annotation in the HOL specification to restrict the record types appropriately.
- *Dealing with “potential” counterexamples.* Heavy use of quantifiers tend to render the underlying SMT solvers incomplete. Isabelle's `safe` tactic might help in cases where such quantifiers can be eliminated.

9. CONCLUSIONS

In this paper, we have described our Isabelle `ismt` tactic. The tactic provides an automated solver for a subset of HOL,

⁷To aid in this process, `ismt` will always print the list of uninterpreted constants that are used in the counterexample.

by translating them appropriately to Yices. As a proof-of-concept work, the `ismt` tactic demonstrates that it is quite feasible to build highly useful interfaces to modern off-the-shelf SMT solvers from within interactive theorem proving assistants.

Not every HOL theorem can be proved by `ismt`. Logics of modern SMT solvers are deliberately weaker, trading expressive power for decidability. Still, we have found that the use of the “uninterpretation” technique to translate literally all HOL theorems to such logics pays-off very nicely in practice. Many tedious theorems can be proven “with the push of a button.”

We also have several items on our Yices wish-list, as well as enhancements we would like to make to `ismt`, as we discuss below.

Quantifier instantiation term patterns. As described in Section 7.2, Yices' in-built quantifier instantiation heuristics did not always work for our use cases. The Simplify and Z3 SMT solvers support quantifier instantiation term patterns, where the user can attach a set of *instantiation patterns* to any quantified assertion [12]. Each instantiation pattern contains a set of term patterns, and a substitution over the term patterns' free variables. If in its proof search the SMT solver finds a set of subterms that jointly matches the term patterns, it will instantiate the quantified assertion according to the substitution and add it to the proof context.

Instantiation patterns can be used to enable systems of rewrite rules by inserting each rewrite as a quantified equality and an instantiation pattern that matches the left-hand-side of each rule. Instantiation patterns can also implement forward- or backward-chaining inference rules, by having the term pattern match either the hypotheses or conclusion of a quantified implication.

For instance, in Section 7.2, we could have given a single instantiation pattern for lemma `str_addrs_simp` that would have solved the subgoal as well as related subgoals:

```
term_patterns: "¬(p' mem str_addrs p h)",
               "str_addrs p n (h(p' := x))"
substitution: p ← p, p' ← p', n ← n, x ← x.
```

which would have simplified the proof significantly.

Abstract counterexamples and call-back tactics. The ability to generate counterexample information is helpful, especially when the counterexamples are small enough (i.e. have only a few free variables and uninterpreted constants). However, we think it would be much more useful to have a Yices command to return an *abstract* counterexample formula, as described in Section 7.2.

Abstract counterexamples would be especially useful in conjunction with Yices' incremental API. In particular, in future work we want to allow the user to attach Isabelle *call-back tactics* that are invoked on the abstract counterexamples produced by Yices. The call-back tactics would then analyze each counterexample and return lemmas that either refute it or infer additional facts such as equalities. These lemmas would then be incrementally asserted into Yices' current proof context the proof search would resume.

This process would continue until Yices is either able to prove the original formula or else produce a counterexample that the call-back tactics could not infer anything about.

Call-back tactics would thus allow the user to safely write domain-specific quantifier instantiation heuristics that could be more powerful than a fixed collection of term patterns.

A further Yices optimization would be to export the E-graph matching API. Users could then register their own term patterns. When a pattern matches during the proof for the first time, Yices would immediately suspend and return references to the matched subterms. There would also have to be commands to query the current proof context (partial model). Exporting the E-graph matcher would also allow call-back tactics to be triggered early, without having to wait for a full counterexample to be built.

Modes of integration. The only mode of integration we have investigated in this work is the oracle mode; where the backend solver and the bridge code is blindly trusted. In order to stay within the pure-LCF style, a proof-generating backend solver, and a proof-replaying (or proof-checking) interface is needed.

Integration with other SMT solvers. The `ismt` tactic has been designed such that other solvers can be plugged in by providing appropriate translators. Currently, we only have a backend for Yices, however. Having translators for multiple backends would make our tactic more useful in the long run, as one can pick the best solver for the task at hand as appropriate. It would of course simplify matters greatly if there was a common SMT language that we could target once and for all. Although SMT-Lib would seem to be a natural target, it currently does not satisfy our needs. It would have to be extended to cover several features (for instance recursive datatypes), as well as support an incremental API and a counterexample format. Grundy et al. provides a nice desiderata for such a specification language [19].

Support for further HOL constants. Currently only a subset of HOL constants are “interpreted,” (those we found useful in our own experience). While this subset is fairly large, supporting more constants would be useful. In particular, devising a more general scheme to translate all non-recursive HOL definitions to Yices equivalents would be desirable. Paired with a simple mechanism to allow users to indicate which constants should be translated, the `ismt` tactic can act as a powerful tool in custom theory development.

Support for further HOL constructs. The translator currently does not support extensible records. Also, rationals and other numeric types (i.e. those other than `int` and `nat`) go uninterpreted during the translation as well. Adding support for these constructs might prove useful in certain application areas.

Parallel proof processing. Our `ismt` implementation does not support Isabelle’s recent multi-core code architecture changes that support parallel proof processing. Thus `ismt` must be called in a single-threaded context. Supporting parallel proofs where each thread might call `ismt` independently would be a nice addition.

Acknowledgments

We would like to thank SRI’s Bruno Dutertre and Natarajan Shankar for answering numerous questions on Yices, Alwen Tiu of Australian National University for discussions on the

`smt` tactic, and Lee Pike for comments on an early draft of our paper.

Availability

Our `ismt` tactic, along with a user’s guide and other supporting material is freely available on the internet with a permissive BSD-style license [5].

10. REFERENCES

- [1] CVC Lite web site. www.cs.nyu.edu/acsys/cvcl/.
- [2] CVC3 web site. www.cs.nyu.edu/acsys/cvc3/.
- [3] haRVey web site. harvey.loria.fr/.
- [4] HOL Lite web site. www.cl.cam.ac.uk/~jrh13/hol-light/.
- [5] The `ismt` tactic web site. www.galois.com/technology/resources/opensource/ismt.
- [6] PVS: Specification and Verification System site. pvs.csl.sri.com/.
- [7] Satisfiability Modulo Theories Solvers. combination.cs.uiowa.edu/smtlib/solvers.html.
- [8] SMT-LIB logics web site. combination.cs.uiowa.edu/smtlib/logics.html.
- [9] SMT-LIB web site. combination.cs.uiowa.edu/smtlib/.
- [10] SMT oracle web site. users.rsise.anu.edu.au/~tiu/smt/.
- [11] Yices web site. yices.csl.sri.com/.
- [12] Z3 web site. <http://research.microsoft.com/projects/z3/index.html>.
- [13] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *PDPAR’03 Workshop, Miami, Florida*, July 2003.
- [14] D. Barsotti, L. P. Nieto, and A. F. Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Electr. Notes Theor. Comput. Sci.*, 145:63–78, 2006.
- [15] B. Dutertre and L. de Moura. The YICES SMT Solver. Available at: yices.csl.sri.com/tool-paper.pdf.
- [16] L. Erkök. Connecting SMT solvers and Isabelle: The `ismt` tactic and the yices bridge. Technical Report, Available upon request from Galois.
- [17] P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. F. Tiu. Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.
- [18] P. Graunke. Verified safety and information flow of a block device. In *Systems Software Verification (SSV’08) Workshop*, Sydney, Australia, Feb. 2008.
- [19] J. Grundy, T. F. Melham, S. Krstić, and S. McLaughlin. Tool building requirements for an API to first-order solvers. *Electr. Notes Theor. Comput. Sci.*, 144(2):15–26, 2006.
- [20] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR’05 Workshop*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, Jan. 2006.
- [21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.