Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

SMT Solvers: New Oracles for the HOL Theorem Prover

Tjark Weber*

Computer Laboratory, University of Cambridge,
15 JJ Thomson Avenue, Cambridge CB3 0FD, U.K.

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper describes an integration of Satisfiability Modulo Theories (SMT) solvers with the HOL4 theorem prover. Proof obligations are passed from the interactive HOL4 prover to the SMT solver, which can often prove them automatically. This makes state-of-the-art SMT solving techniques available to users of the HOL4 system, thereby increasing the degree of automation for a substantial fragment of its logic. We compare a translation to Yices's native input format with a translation to SMT-LIB format.

1 Introduction

Interactive theorem proving is a well-established approach for formal verification of hardware and software. It can show system correctness with mathematical certainty, thereby providing extremely high reliability assurances. Interactive theorem provers such as Coq [5], HOL4 [33], Isabelle [26] and PVS [30] typically support rich specification languages, e.g., higher-order logic or dependent type theory, enriched with facilities to define algebraic data types, record types, recursive functions, etc. [28] Unfortunately, interactive theorem proving is notoriously labor intensive, thus costly and limited in its industrial application.

On the other hand, much research has been devoted to the development of automated reasoning algorithms. Once considered the holy grail of artificial intelligence, practically successful automated theorem provers are now available for a variety of logics, perhaps most notably for first-order [34] and propositional logic [37]. Consequently, researchers have on many occasions proposed to integrate automated provers with interactive theorem provers [19,31,36], thereby increasing the degree of automation available in interactive theorem proving and ultimately enhancing its applicability.

In the past years, automated theorem provers have emerged that support fragments of first-order logic with equality and combinations of various (usually decidable) background theories, e.g., linear arithmetic or theories of data structures such as lists, arrays and bit vectors [21]. These provers are called *Satisfiability Modulo Theories* (SMT) solvers. SMT solvers such as Yices [12] have numerous applications, e.g., in model checking, scheduling and compiler optimization. They are also of particular value in formal verification, where specifications and verification conditions can often be expressed as SMT formulas.

We present an integration of SMT solvers with the HOL4 theorem prover. Proof obligations that arise in HOL4 are translated from higher-order logic (see Sect. 2) into the SMT solver's input language. We have implemented translations into Yices and SMT-LIB 1.2 [32] format. In particular the Yices translation can handle a substantial fragment of higher-order logic, including propositional connectives, uninterpreted functions, various arithmetic and bit-vector operations, but also quantifiers and λ-terms. The SMT-LIB translation is more restrictive, but provides support for a large number of SMT solvers, e.g., CVC3 [3] and Z3 [11]. Both translations are described in detail in Sect. 3. We briefly present some experiments in Sect. 4, before discussing related work in Sect. 5. Section 6 concludes.

2 Background

The HOL4 system is an interactive theorem prover. Like Isabelle/HOL [26], HOL Light [15], and several other theorem provers for higher-order logic, it is based on Church's simple theory of types [8]. Thus, its logic is

* This work was supported by EPSRC grant EP/F067909/1.

typed. Types σ are given by the following grammar:

$$\sigma ::= \alpha \mid (\sigma_1, \ldots, \sigma_n)c$$

Here α ranges over an infinite set of *type variables*, and c ranges over *type constructors* (of arity $n \geq 0$). Type constructors include bool (of arity 0) and the function space constructor $\rightarrow$ (of arity 2). The latter is usually written in infix notation and associates to the right: e.g., $\alpha \rightarrow \beta \rightarrow \gamma$ is short for $(\alpha, (\beta, \gamma)\rightarrow)\rightarrow$.

Terms t are explicitly typed. They are given by the following grammar:

$$t ::= x_\sigma \mid c_\sigma \mid (t_{\sigma\rightarrow\sigma'}\, t_\sigma)_{\sigma'} \mid (\lambda x_\sigma.\, t_{\sigma'})_{\sigma\rightarrow\sigma'}$$

Thus, a term is either an (explicitly typed) variable, a constant, an application, or a λ-abstraction. Constants include equality, propositional connectives, quantifiers, etc. We frequently omit type annotations and unnecessary parentheses when they can be deduced from the context, and we assume that application is left-associative: e.g., $f\, x\, y$ is short for $((f\, x)\, y)$.

The HOL4 system is implemented in Standard ML, a typed functional programming language with an advanced module system [24]. Types and terms of higher-order logic are implemented as Standard ML data types.[1] Formulas are simply terms of type bool.

Formulas are stated as conjectures by the user. They become proof obligations that are then proved through a series of *tactic* invocations. Tactics are functions written in Standard ML that, when applied to a proof obligation, yield a (possibly empty) list of new proof obligations. Thus, proving $\vdash \varphi \wedge \psi$, for instance, can be reduced to proving $\vdash \varphi$ and proving $\vdash \psi$. Of course tactics can be much more powerful than this trivial example shows, implementing complex decision procedures or rewriting strategies.

Users of the HOL4 system can define their own tactics. This paper describes a tactic that, given a proof obligation φ in HOL4, translates $\neg\varphi$ into the input language of an SMT solver, invokes the solver, and if the SMT solver determines the negation to be unsatisfiable, establishes φ as a theorem in HOL4. The overall architecture is shown in Fig. 1. Communication with the SMT solver is via files. Several leading SMT solvers provide C APIs that could be used for a more direct integration, but only at the cost of portability: foreign function interface libraries like Huelsbergen's [18] that achieve interoperability between Standard ML and C are compiler-specific.

The resulting theorem is tagged with an "oracle" string to indicate that its derivation was not checked by HOL4's own inference kernel. The HOL4 system stands in the tradition of Milner's LCF theorem prover [14] and implements theorems as an abstract data type. New theorems can be obtained only through a fixed set of operations on this data type, corresponding to the axiom schemata and inference rules of higher-order logic. This design greatly reduces the trusted code base: potentially complex decision procedures cannot derive inconsistent theorems, provided the theorem data type itself is implemented correctly.

In contrast, the SMT-based tactic presented here requires that one trusts the SMT solver and our interface code to produce correct results. A bug in the SMT solver (or in the interface) could potentially lead to inconsistent theorems in the HOL4 system—tagged, however, with said oracle string. More recently, we have addressed this issue by implementing *proof reconstruction* [6, 7]: proofs of unsatisfiability found by the SMT solver Z3 are translated into HOL4 inferences, thereby eliminating the oracle tag and deriving a theorem with the same degree of confidence that is attributed to any other theorem proved in the HOL4 system proper.

3 Translation

HOL4 supports higher-order logic, whereas SMT solvers typically support (fragments of) first-order logic over various background theories, e.g., linear arithmetic, arrays, bit vectors. Thus, proof obligations from the HOL4 system must be translated into the input language of the SMT solver. We have implemented translations into Yices's native input format [12], and into SMT-LIB format [32]. Their description constitutes the main part of this paper. We use *italics* for variables of higher-order logic, sans-serif for specific type constructors and constants, and `typewriter` font for SMT solver input.

The translations are comparatively simple. We only aim to support the fragment of higher-order logic that is directly backed by SMT solvers. No sophisticated encoding of higher-order into first-order logic is performed, and since SMT solvers support sorted first-order logic, no encoding of type information into terms is necessary. Thus, many of the complications that are dealt with in Meng's and Paulson's integration of first-order provers with Isabelle/HOL [23] do not arise. However, the translations are not quite as trivial as the grammars for types and terms given in Sect. 2—with just two and four cases, respectively—might suggest. Our goal is to translate HOL4's types and constants to *semantically corresponding* types and constants in the SMT world. Therefore, different type constructors and constants cannot be treated uniformly. Also, there is the difficulty of translating polymorphism, which is not supported by Yices or SMT-LIB: we use a technique known as monomorphization (see Sect. 3.3.8).

[1] Extending the shown term grammar, HOL4 internally uses de Bruijn indexes and explicit substitutions to represent λ-terms. Externally, however, the interface is to a name-carrying syntax.

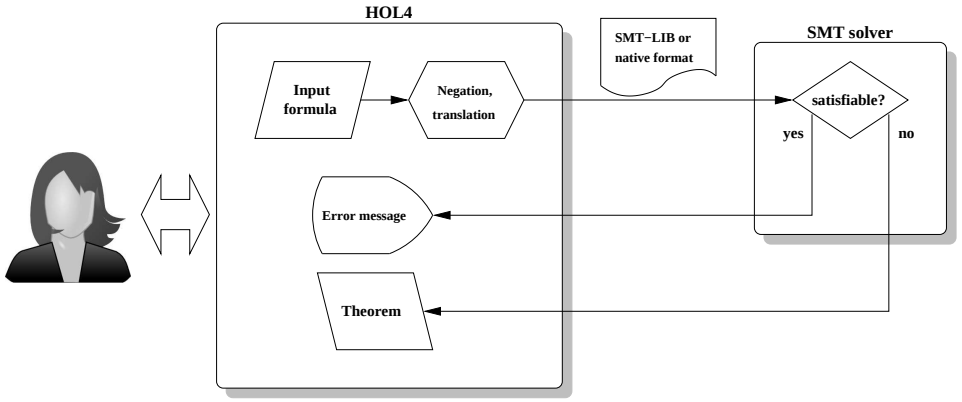

Fig. 1. System Architecture

3.1 Abstraction

Both our Yices and our SMT-LIB translation employ abstraction as a means of dealing with terms and types that are not supported by the respective target format. This technique, called *uninterpretation* in [13], replaces all offending sub-terms by uninterpreted constants (of the same type as the sub-term). Likewise, offending types are replaced by type variables. These are translated as new "basic" (i.e., uninterpreted) types in Yices, and as "extra sorts" in the SMT-LIB format.

As a very simple example, consider $P_{\alpha\to\mathsf{bool}}\ x_\alpha$, where P and x are variables, and α is a type variable: after negation, this formula is translated as

```
(define-type a)
(define P::(-> a bool))
(define x::a)
(assert (not (P x)))
```

in Yices, and as

```
:extrasorts (a)
:extrafuns ((x a))
:extrapreds ((P a))
:formula (not (P x))
```

in SMT-LIB format.[2] The resulting problem is easily determined to be satisfiable by Yices; thus, no theorem is derived in this case.

Multiple occurrences of a term (or type) are of course replaced by the same uninterpreted constant (or sort, respectively) each time. To this end, two dictionaries are built during the translation that map types and terms to their abstraction. Terms are treated modulo α-equivalence.

The original formula is an instance of its abstraction. Therefore, unsatisfiability of the abstracted formula implies unsatisfiability of the original. Hence abstraction is sound, in the sense that it does not lead to inconsistent theorems. However, it introduces incompleteness: the SMT solver may fail to prove certain theorems and return spurious countermodels. This is true of Yices anyway when the input contains λ-expressions or quantifiers, so we consider it a minor issue for the time being.

3.2 SMT-LIB

The SMT-LIB language [2,32] is the de facto standard for the input format of SMT solvers. It is supported by all major SMT solvers. Benchmarks used for the annual Satisfiability Modulo Theories Competition (SMT-COMP) are encoded in SMT-LIB format.

The SMT-LIB benchmark collection is structured in a modular fashion, based on various *theories* and *logics* that share a common Lisp-like syntax, but differ in their signatures (i.e., built-in sorts, functions and predicates) and axiomatizations, as well as in syntactic restrictions. The SMT-LIB format is significantly less expressive than the native input formats of various SMT solvers: e.g., data types and λ-terms are not supported directly, and we make no attempt to encode them. On the other hand, the SMT-LIB format provides support for a vast number of SMT solvers at once.

We translate to a logic called AUFLIRA: closed linear formulas with free function and predicate symbols over a theory of arrays of arrays (i.e., two-dimensional arrays) of integer index and real value.[3] Clearly, one could also implement translations to other SMT-LIB sublogics, or dynamically choose an appropriate logic (and translation) based on the input formula.

The SMT-LIB language is first-order. Moreover, SMT-LIB 1.2 makes a clear distinction between terms and formulas. In HOL4, formulas are themselves terms (of type bool); in SMT-LIB 1.2, this is not the case. Our translation therefore employs two functions: one to translate formulas, another one to translate terms. These functions are mutually recursive according to the production rules of the SMT-LIB grammar. (Very recently, version 2.0 of the SMT-LIB language was released [2]. It no

[2] We omit additional header data from the presentation that is required to obtain a syntactically valid SMT-LIB file.

[3] In our translation, we do not make use of arrays yet.

longer makes this distinction between terms and formulas in its grammar, although it is still first-order. While we are currently updating our implementation to target SMT-LIB 2.0, this paper still describes the SMT-LIB 1.2 translation. Most translation challenges remain the same between the two versions of the SMT-LIB language.)

3.2.1 Types

The only supported types are int and real. They are translated to their SMT-LIB counterparts `Int` and `Real`, respectively. All other types are left uninterpreted, i.e., declared as extra sorts. This also applies to type bool when it is used as the type of a term that (according to the SMT-LIB grammar) is not a formula.

3.2.2 Basics

The propositional constants T, F and connectives $\Longleftrightarrow$, $\Longrightarrow$, $\vee$, $\wedge$ and $\neg$ are translated to their SMT-LIB counterparts, as is equality. ($\Longleftrightarrow$ and $=$ are the same constant in HOL4, distinguished just by their type: the former is equality on type bool.) HOL4's if-then-else constructs if c then t_1 else t_2 and bool_case t_1 t_2 c are translated as `if_then_else c` $\mathtt{t_1}$ $\mathtt{t_2}$ if they occur at the formula level (with t_1 and t_2 of type bool), and as `ite c` $\mathtt{t_1}$ $\mathtt{t_2}$ if they occur as a term.

The Lisp-like syntax of SMT-LIB generally allows for a simple recursive structure of our translation. A curried application $f\,x_1\,\ldots\,x_n$ is typically translated by looking up the SMT-LIB function `f` corresponding to the HOL operator f, translating each argument term x_i, and then putting the results together to obtain a single string `f` $\mathtt{x_1}$ $\ldots$ $\mathtt{x_n}$. This works well for most propositional and arithmetic operators. We perform η-expansion to eliminate partial applications where necessary.

3.2.3 Arithmetic

Integer and real-number numerals are translated into SMT-LIB notation, as are negation, addition, subtraction, multiplication on integers and reals. Comparison operators $<$, $\leq$, $>$, $\geq$ on these types are also supported.

For certain other functions, e.g., min, max and abs, which are not available in SMT-LIB directly, we introduce suitable definitions. For instance, the proof obligation abs $x_{\mathsf{int}} \geq 0$ is translated as

```
:extrafuns ((hol_int_abs Int Int) (x Int))
:assumption (forall (?x Int)
    (= (hol_int_abs ?x)
        (ite (< ?x 0) (- 0 ?x) ?x)))
:formula (not (>= (hol_int_abs x) 0))
```

This is readily determined to be unsatisfiable by several SMT solvers.

3.2.4 Quantifiers

HOL4's quantifiers, $\forall$ and $\exists$, are translated to their SMT-LIB counterparts, `forall` and `exists`. For instance, the "drinker paradox", $\exists x_\alpha.\,(P\,x \implies \forall y_\alpha.\,P\,y)$, is translated as

```
:extrasorts (a)
:extrapreds ((P a))
:formula (not (exists (?x a)
    (implies (P ?x) (forall (?y a) (P ?y)))))
```

which is of course unsatisfiable. Only first-order quantification is supported; higher-order quantification is abstracted away.

The treatment of bound variables requires some care. Unlike other variables that occur in the input formula, they must not be declared as (nullary) extra functions. Their types, however, must be declared as extra sorts, and last not least, their names must begin with a question mark in concrete SMT-LIB syntax.

Simultaneous quantification over several variables, as in $\forall x_\alpha\,y_\beta.\,P\,x\,y$, in HOL4 is just syntactic sugar for nested quantifiers, i.e., $\forall x_\alpha.\,\forall y_\beta.\,P\,x\,y$. Nevertheless we translate this formula by making use of a corresponding feature in SMT-LIB (and likewise in Yices) that allows simultaneous quantification.

3.2.5 Let expressions

Let expressions, e.g., let $x = 1$ in $x > 0$, in HOL4 are syntactic sugar for LET $(\lambda x.\,x > 0)$ 1, where LET is a polymorphic constant of type $(\alpha \to \beta) \to \alpha \to \beta$. Let expressions at the formula level are translated to their SMT-LIB counterparts: e.g., the above becomes

```
:formula (not (let (?x 1) (> ?x 0)))
```

When the abbreviated expression is of type bool, SMT-LIB's `flet` construct is used instead.

Note that SMT-LIB, unlike HOL4, does not support let expressions inside terms. Therefore, we compile the latter away: LET $M\,N$ in HOL4, by definition, is $M\,N$ (i.e., M applied to N). This is followed by β-reduction if possible. Alternatively, one could lift let expressions outward: φ(let $x = e$ in t) is equivalent to let $x = e$ in $\varphi(t)$, provided e does not contain any variables bound in φ and x does not occur in φ except in t.

3.2.6 Anonymous and Higher-Order Functions

The SMT-LIB format does not support λ-abstractions. They are therefore replaced by uninterpreted constants, as described in Sect. 3.1, if they are not eliminated by β-reduction. Moreover, higher-order functions or predicates are not supported. Therefore, even the function space constructor, $\to$, is subject to abstraction when it occurs in the argument type of a function or predicate.

3.3 Yices

We chose Yices because it is one of the leading SMT solvers [1], with an expressive and well-documented input language. Yices has been developed cognizant of deduction in PVS, and its input language contains features (e.g., λ-abstraction) not found in most other SMT solvers. We use version 1.0.23 of Yices, its latest release at the time of writing.

3.3.1 Types

Supported ground types include bool, num (HOL4's type of natural numbers, i.e., non-negative integers), int and real. These are translated to their Yices counterparts `bool`, `nat`, `int` and `real`, respectively. First-order and higher-order functions are supported, as are tuples, records, fixed-width bit-vector types and user-defined data types, e.g., lists (see below).

3.3.2 Basics

Our Yices translation supports the same propositional connectives as the SMT-LIB translation, i.e., T, F, $\iff$, $\implies$, $\vee$, $\wedge$ and $\neg$, as well as equality and if-then-else. These are translated to their Yices counterparts `true`, `false`, `=`, `=>`, `or`, `and`, `not`, as well as `=` and `ite`.

Yices has a Lisp-like input syntax, similar to SMT-LIB. However, just like in HOL4, there is no distinction between formulas and terms. Therefore, a single translation function suffices to translate both. `ite` is used for if-then-else (regardless of whether it occurs in a formula or a term), and `=` is used both for equality and equivalence.

3.3.3 Arithmetic

In addition to the arithmetic types and operators that are supported by our SMT-LIB translation (i.e., addition, subtraction, multiplication on int and real, comparison operators, and min, max, abs), the Yices interface supports natural numbers: numerals of type num, arithmetic and comparison operators on naturals, the successor function (SUC x_{num} is translated as `+ 1 x`). Also integer division, div, and the modulo operation, mod, are supported (both on integers and naturals), as is division of real numbers, /.

Because of semantic differences between some of the latter operations in HOL4 and in Yices, the translation does not merely consist of a one-to-one mapping from HOL4 constants to (allegedly) corresponding built-in Yices constants. Instead, we typically have to define corresponding constants in Yices ourselves. This is further discussed in Sect. 3.4 below.

3.3.4 Quantifiers

Yices—and hence our translation—supports universal and existential quantifiers of arbitrary order. They are translated to their Yices counterparts, `forall` and `exists`. As an example, consider the following (not universally valid) higher-order formula: $\forall f_{\alpha\to\beta}.\, \exists g_{\beta\to\alpha}.\, \forall x_\alpha.\, g\,(f\,x) = x$. This is translated as

```
(define-type a)
(define-type b)
(assert (not (forall (f::(-> a b))
    (exists (g::(-> b a))
        (forall (x::a) (= (g (f x)) x))))))
```

which Yices currently reports as "unknown". Generally Yices is incomplete when quantifiers are used, and the model that is returned for a problem with "unknown" status may be spurious. However, Yices will not erroneously claim unsatisfiability; thus, soundness is not compromised.

3.3.5 Let expressions

In contrast to the SMT-LIB translation, our Yices interface allows let expressions to occur anywhere, also inside terms. let expressions are translated to their Yices counterparts: e.g., let $x = 1$ in $x > 0$ becomes

```
(assert (not (let ((x 1)) (> x 0))))
```

Yices has no `flet` construct; since there is no distinction between formulas and terms, `let` is also used for terms of type bool.

3.3.6 Anonymous and Higher-Order Functions

Yices provides a `lambda` construct, which is used to translate λ-abstractions. We first perform β-normalization in HOL4: although this could in theory cause a blowup of the formula, we generally expect the HOL system to be better at dealing with λ-terms than the SMT solver. This is followed by η-expansion, because we found that Yices only provides partial support for partial function applications.

We do not make use of Yices's ability to bind several variables simultaneously. In HOL4, $\lambda x_\alpha\, y_\beta.\, f\,x\,y$ is merely syntactic sugar for $\lambda x_\alpha.\, \lambda y_\beta.\, f\,x\,y$, and the function space constructor, $\to$, always takes two arguments. Consequently, we never use function types with more than one domain (which are supported by Yices). Instead, we use currying when translating functions with multiple arguments. Thus, assuming the return type of f is translated as `c`, the type of the above λ-term will be translated as `-> a (-> b c)`, rather than `-> a b c`.

Update of a function f at position a with value b in HOL4 is written $(a =+ b)\, f$. Thus, the update operator, $=+$, is of polymorphic type $\alpha \to \beta \to (\alpha \to \beta) \to \alpha \to \beta$. It is translated to its Yices counterpart: the above expression becomes `update f (a) b`.

3.3.7 Tuples

Product types $\alpha \times \beta$ are mapped to their Yices counterparts, `tuple a b`. Tuples are created by the comma operator in HOL4; (x, y) is translated as `mk-tuple x y`. Accessor functions for a tuple's components, FST p and SND p, are translated as `select p 1` and `select p 2`, respectively.

In HOL4, tuples with more than two components are supported through right-nesting: e.g., $(x_\alpha, y_\beta, z_\gamma)$ is merely syntactic sugar for $(x_\alpha, (y_\beta, z_\gamma))$ of type $\alpha \times (\beta \times \gamma)$. For ease of implementation we use the same technique when translating to Yices, and do not make use of tuples with more than two components.[4]

3.3.8 Records

Record types in HOL4 are semantically equivalent to product types, but with named field access and update. For types that were introduced by a record type definition, such as

```
Hol_datatype
  'person = <| employed : bool ; age : num |>'
```

a corresponding type definition is generated in Yices:

```
(define-type person
    (record employed::bool age::nat))
```

Our Yices interface at the moment does not support recursive record types.

In HOL4, record types can be parameterized, i.e., depend on type arguments. Since Yices only supports monomorphic types (over uninterpreted basic types), we may need to create multiple copies of a parameterized record type. For instance, if (α)foo was introduced by a record type definition

```
Hol_datatype 'foo = <| bar : 'a |>'
```

an occurrence of both (α)foo and (β)foo in the input formula leads to *two* type definitions

```
(define-type a)
(define-type foo1 (record bar1::a))
(define-type b)
(define-type foo2 (record bar2::b))
```

In contrast to [13], where this process is called *monomorphisation*, we do not mingle string representations of its argument types into the record type name, because this makes it tricky to guarantee unique names. Instead, our implementation uses freshly generated identifiers (see Sect. 3.4).

Field selectors in HOL4 are functions named <record_type>_<field>. Field access can be written with a dot, as in x.age; however, assuming x is of type person, this is merely syntactic sugar for person_age x. It is translated to Yices as `select x age`.

Field updates, as in x with employed := T, are syntactic sugar for more general functional field updates. Provided x is of type person, this particular update corresponds to person_employed_fupd (K T) x, where K is the usual combinator that drops its second argument, returning its first. Since the employed field in our example is of type bool, the update function person_employed_fupd is of type (bool → bool) → person → person. We translate the general form of a functional field update, <record_type>_<field>_fupd f x, by first translating f (<record_type>_<field> x), i.e., f applied to the current field value, performing β-reduction in HOL4 if possible. Call the resulting string `fx`, then the functional update is translated as `update x <field> fx` in Yices.

Record literals, such as <| employed := F ; age := 65 |>, in HOL4 are syntactic sugar for a sequence of field updates to an arbitrary (i.e., uninterpreted) initial record. We translate them just like this, and do not use Yices's `mk-record` feature at the moment.

3.3.9 Data types

HOL4 supports nested and mutually recursive data types. Popular data types include the one-element type one, disjoint sum types (with constructors INL and INR), the option type (with constructors NONE and SOME), the list type (with constructors NIL and CONS).

Our Yices interface supports recursion only as long as it is not nested or mutual. (Thus, all of the abovementioned types are fully supported.) When a data type[5] is encountered in the input formula, its definition is translated to a corresponding data type definition in Yices. For instance, the HOL4 definition of the (α)list type,

```
Hol_datatype 'list = NIL | CONS of 'a => list'
```

becomes

```
(define-type a)
(define-type list (datatype NIL
    (CONS hd::a tl::list)))
```

Note that HOL4 does not provide accessor functions. Thus, their names (`hd`, `tl` above) do not occur elsewhere and can be chosen freely, as long as they do not clash with other identifiers. The discussion about multiple copies of parametric record types (see the previous subsection) also applies to parametric data types. We do not use Yices's "scalar" types: enumeration types, with nullary constructors only. These are just a special case of data types.

Yices automatically declares a recognizer `C?` for every data type constructor `C`. Recognizers are predicates on the data type; recognizer `C?` returns true for a given element e of the data type if e is of the form `(C ...)`.

[4] Taking full advantage of Yices's input language could potentially improve the solver's performance on problems that involve tuples with many components. However, at present there are no such problems in the HOL4 library.

[5] Other than bool and num, which can also be viewed as data types, but are supported as basic types by Yices.

We use recognizers to implement case distinction: e.g., list_case $b\ f\ l$, the case constant on lists, is translated as `ite (NIL? l) b (f (hd l) (tl l))`. Case distinctions on data types with three or more constructors lead to cascaded if-then-else expressions. Note the use of accessor functions to obtain a constructor's arguments.

Support for recursive functions is preliminary. Simple proof obligations, e.g., length NIL $= 0$, can be shown by unfolding their definition. More generally, we can add the (universally quantified) recursive definition as an antecedent to the proof obligation. The use of quantifiers, however, introduces incompleteness, and in our experience Yices will often fail to solve the problem.

Note that overloading in HOL4 is achieved merely by pretty-printing, but does not affect the underlying logic, and polymorphic recursion is not supported at all.

3.3.10 Bit vectors

HOL4 provides fixed-width bit-vector types, e.g., word8, word16, word32.[6] These are translated to their Yices counterparts: as `bitvector 8`, `bitvector 16`, `bitvector 32`, etc.

Our translation supports a whole range of bit-vector operations. Literals, such as 0w and 1w, are translated using Yices's `mk-bv` function. Bit-vector concatenation, @@, and bit-vector extraction, ><, are mapped to `bv-concat` and `bv-extract`, respectively. Other operations that are supported by mapping them directly to built-in Yices functions include left and right shift, bitwise conjunction, disjunction, negation (one's complement) and xor, bit-vector addition, subtraction, multiplication and negation (two's complement), signed and unsigned comparison.

HOL's w2w function, which converts between bit vectors of different width, is translated using either `bv-extract` or `bv-concat`, depending on the width of its argument and result. More precisely, $(\mathsf{w2w}\ x_{\mathsf{word}m})_{\mathsf{word}n}$ is translated as `bv-extract (n-1) 0 x` if $n \leq m$, and as `bv-concat (mk-bv (n-m) 0) x` otherwise. Because we are dealing with fixed-width bit vectors only, the values of n and m, and hence of $n-1$ and $n-m$, are known at translation time.

Extracting a single bit from a bit vector, denoted by an infix $'$ in HOL4, is translated using Yices's `bv-extract` function. However, the result of `bv-extract` is again a bit vector (in this case of length 1), while the $'$ operator in HOL4 returns a Boolean. Therefore, a final comparison with `0b1` ensures type correctness: $x_{\mathsf{word}m}\,'\,n$ is translated as `= (bv-extract n n x) 0b1`, provided n is a numeral less than m. HOL4's word_msb function, which extracts a bit vector's most significant byte, is translated using a similar approach.

[6] These are implemented as instances of a polymorphic (α)word type, based on [16]. Our Yices translation only supports fixed-width instances of this type.

3.4 Caveats

Because a bug in our translation could lead to inconsistent theorems in the HOL4 system (albeit tagged with an "oracle" string), it is imperative to get things right. Arguably, the translation is quite straightforward in principle: many HOL constants are merely replaced by corresponding constants in the SMT solver's input language, using a simple dictionary approach. However, we have identified three potential pitfalls that deserve to be pointed out.

Identifiers. When names of HOL variables or constants are re-used in the SMT solver's input, we must ensure that they do not accidentally coincide with identifiers or keywords that have special meaning to the SMT solver (e.g., names of built-in functions). We must also ensure that these names are in fact regarded as valid identifiers by the SMT solver; they must not contain special characters (e.g., white-space) that would render them illegal. Note that we cannot rely on HOL4's built-in parser to rule out such identifiers: the HOL4 prover grants its users direct access to the underlying Standard ML prompt, which allows to use any string as the name of a constant, variable or type. Moreover, identifiers for fresh variables introduced by abstraction (see Sect. 3.1) or for type copies and accessor functions (see Sect. 3.3) must not coincide with any other identifiers.

Instead of re-using HOL identifiers, a conceptually simpler solution—and the one that we take in our current implementation—is to uniformly generate a fresh name for every HOL identifier. This makes it easy to guarantee uniqueness and validity, thereby elegantly solving most of the technical issues just described (at the cost of readability of the resulting SMT input file, which, however, is not intended for human eyes anyway).

Semantic differences. Replacing a HOL constant by its "obvious" SMT counterpart may not be the right thing to do. There are subtle semantic differences between certain HOL and SMT functions that would render this approach unsound. For instance, subtraction $m - n$ on naturals in HOL4 is defined to be 0 if $n > m$; in Yices, however, the result would be a negative integer. Our Yices translation therefore introduces a wrapper function `hol_num_minus` for subtraction on naturals, which we define (in Yices) as follows:

```
(define hol_num_minus::(-> nat nat nat)
    (lambda (x::nat y::nat)
        (ite (< x y) 0 (- x y))))
```

Other examples are $x \,\mathsf{div}\, 0$ and $x \,\mathsf{mod}\, 0$, which are unspecified (but well-formed) in HOL4, whereas both are not type-correct in Yices. We use the following wrapper function `hol_num_div` to translate HOL's div operator:

```
(define hol_num_div0::(-> nat nat))
(define hol_num_div::(-> nat nat nat)
```

```
(lambda (x::nat y::nat) (ite (= y 0)
    (hol_num_div0 x) (div x y))))
```

Note the use of an unspecified function `hol_num_div0`. The translation of `div` and `mod` on integers is handled similarly.

Error checking. It would be convenient if our translation could generate potentially ill-formed SMT input (e.g., containing non-linear arithmetic terms), and rely on the SMT solver for proper error checking. Yices, however, "does no checking and can behave unpredictably if given bad input" [12]. One can enable a certain amount of type checking in Yices via a command-line switch, but if we want to ensure soundness, the burden to produce correct input for the SMT solver is essentially on our translation.

4 Experiments

We have applied Yices to a range of "typical" proof obligations from the HOL4 library, involving data types, quantifiers, etc., as well as to several sample proof obligations arising from work on machine-code verification [25]: quantifier-free formulas that involve bit-vector operations and linear integer arithmetic. These proof obligations had been proved interactively in HOL4 before. Based on our experiments, our key experiences are:

- The SMT-LIB interface, due to its restrictions, does not add very much to existing proof procedures. In particular, automated term rewriting and decision procedures for linear arithmetic have been available in HOL4 for years [27], and their performance is typically satisfactory for the proof obligations that arise in interactive verification.
- Yices performs very well for proof obligations that involve bit-vector operations and linear arithmetic only. It was able to prove all our sample formulas, some of which had rather involved interactive proofs, in split seconds. Size and logical complexity did not pose a serious challenge to the solver; the proof obligations that arise in interactive verification are typically much smaller than what SMT solvers are designed to handle.
- Support for quantifiers and λ-terms, however, could be improved. Yices typically gave inconclusive answers for formulas containing nested quantifiers or higher-order functions in a non-trivial way. Perhaps term patterns (i.e., triggers), as supported by Z3, could help improve Yices's heuristics for quantifier instantiation.

We expect to gather more practical experience with our SMT solver interface as we carry out larger case studies, and—since our implementation is now part of the standard HOL4 distribution—through feedback from other HOL4 users. A more detailed quantitative evaluation is beyond the scope of this paper.

5 Related Work

There is a substantial amount of related work. SMT solvers have been a hot research topic for the past few years, and an integration with interactive theorem provers has been pursued by several researchers.

Perhaps most closely related is a recent integration of Yices with the Isabelle/HOL system by Erkök and Matthews [13]. Aside from targeting a different theorem prover, we have not only implemented a translation to Yices, but also a (less expressive) translation to SMT-LIB format, and our Yices interface supports bit-vector operations.

Barsotti et al. [4] describe a verification case study in Isabelle/HOL that benefitted from the automation of ICS and CVC Lite (predecessors of Yices and CVC3, respectively). Their translation uses the SMT-LIB language, similar to our SMT-LIB interface. Yices is also available as a decision procedure in recent versions (4.0 and higher) of PVS [29]. Since Yices and PVS share a common type system, the translation is probably much more direct than ours.

Collavizza and Gordon [9] recently integrated Yices with HOL4 to solve verification conditions that arise from symbolic execution of Java programs. Their interface was tailored toward this particular application. It only supported a small subset of the HOL terms that our translation can handle. Our interface has since been integrated with their verification condition generator, where it makes a fine replacement for their (now obsolete) more restrictive interface.

Hurlin et al. [20] have integrated the SMT solver haRVey with Isabelle/HOL in a proof-producing fashion. Their work, however, is restricted to formulas of first-order logic (including quantifiers, uninterpreted functions and equality). Linear arithmetic or other background theories are not supported.

The integration of CVC3 with HOL Light by McLaughlin et al. [22], which is also proof-producing, supports (a subset of) the AUFLIA logic of SMT-LIB, i.e., closed linear formulas over the theory of integer arrays with free sort, function and predicate symbols. Conchon et al. [10] present an integration of their solver Ergo with Coq. Ergo supports polymorphic first-order logic, which simplifies the translation from Coq, and generates proofs for uninterpreted functions and linear arithmetic.

Very recently, Böhme [6] and Weber [7] have presented a more comprehensive integration of Z3 with Isabelle/HOL and HOL4. Their focus is on proof reconstruction (i.e., translating Z3's proofs into Isabelle/HOL inferences), rather than on a translation from Isabelle/HOL to Z3.

6 Conclusions

We have presented an integration of Yices and other SMT solvers, e.g., CVC3 and Z3, with the HOL4 theorem prover. In contrast to related work, we support both i) Yices's native input format, and ii) the SMT-LIB format. This paper contained a head-to-head comparison of both translations. Yices has a rich input language, of which we used almost every feature (with the notable exception of subtypes and dependent types); consequently, the Yices translation can handle a substantial fragment of higher-order logic. The SMT-LIB translation is more restrictive, but provides support for a large number of SMT solvers. Our implementation has become part of the Kananaskis 5 release of HOL4 and can be obtained from its Subversion repository [17].

One could work around many of SMT-LIB's restrictions: higher-order logic could be encoded in first-order logic (through the use of a binary application operator, as in [23]), natural numbers could be treated as integers with an additional non-negativity constraint, data types could be characterized by first-order axiomatizations. However, this would add significant complexity to our (otherwise fairly direct) translation, and SMT solver support for the resulting encodings would likely be rather inefficient: for instance, encoding application would interact badly with the congruence-closure algorithm often used in SMT solvers. Since various SMT solvers already provide direct support for these features, it seems more worthwhile to implement custom translations, as we have done for Yices.

Still, the fact that there is no single, solver-independent input language that covers these features seems unfortunate. We believe that it would be desirable to have more expressive power in the SMT-LIB language itself. The SMT-LIB benchmark collection is already organized in a modular and extensible way, and since features like natural numbers or data types are supported by various SMT solvers, adding syntax for them would seem beneficial to us, even if no benchmark problems make use of them yet.

Our work targeted version 1.2 of the SMT-LIB standard. Very recently, version 2.0 was released [2]. The revised standard allows for some simplifications, e.g., because it identifies formulas with terms of type `bool`. On the other hand, certain useful features (in particular algebraic data types) are still missing. We are currently updating our implementation to target SMT-LIB 2.0; most translation challenges remain the same between the two versions of the language.

Our translations are largely straightforward. Nevertheless, it is tricky to get every detail right. Small mistakes that introduce unsoundness are made all too easily. To address this issue, we have (in separate work) implemented proof reconstruction: proofs of unsatisfiability produced by the SMT solver Z3 are translated into a fixed set of inference rules provided by the HOL4 theorem prover [6, 7].

We only translate single proof obligations (including a list of assumptions) at the moment. Other context information, e.g., definitions and axioms of the theory under consideration, must be added by the user manually. A more convenient approach, suggested in [35], would be to identify relevant axioms (and perhaps helpful lemmas) automatically, and pass them to the SMT solver as additional assumptions, along with the proof obligation itself.

Last, but not least, we plan to make better use of models found by the SMT solver (by displaying them to the user of the HOL4 system as a potential counterexample), and to add support for further HOL constructs.

Acknowledgments

The author is indebted to Hasan Amjad and Michael Gordon, without whom this paper would not exist, to Magnus Myreen for providing various proof obligations involving bit vectors, and to the anonymous referees who commented on an earlier version.

References

1. Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 4th Annual Satisfiability Modulo Theories Competition (SMT-COMP '08). To appear.
2. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0, August 2010. Available from `http://combination.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.08.28.pdf`. Retrieved October 22, 2010.
3. Clark Barrett and Cesare Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Proceedings*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
4. Damian Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Electronic Notes in Theoretical Computer Science*, 145:63–78, 2006.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer, 2004.
6. Sascha Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In *7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, 2009.
7. Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Proceedings*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.
8. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
9. Hélène Collavizza and Mike Gordon. Integration of theorem-proving and constraint programming for

software verification. Technical report, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis, November 2008.
10. Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight integration of the Ergo theorem prover inside a proof assistant. In *AFM '07: Proceedings of the Second Workshop on Automated Formal Methods*, pages 55–59. ACM Press, 2007.
11. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
12. Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Available from `http://yices.csl.sri.com/tool-paper.pdf`. Retrieved October 22, 2010.
13. Levent Erkök and John Matthews. Using Yices as an automated solver in Isabelle/HOL. In *AFM '08: Proceedings of the Third Workshop on Automated Formal Methods*, pages 3–13. ACM Press, 2008.
14. Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.
15. John Harrison. The HOL Light theorem prover. Available from `http://www.cl.cam.ac.uk/~jrh13/hol-light/`. Retrieved October 22, 2010.
16. John Harrison. A HOL theory of Euclidean space. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Proceedings*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005.
17. HOL 4 Kananaskis 5. Available from `http://hol.sourceforge.net/`. Retrieved October 22, 2010.
18. Lorenz Huelsbergen. A portable C interface for Standard ML of New Jersey, January 1996. Available from `http://www.smlnj.org/doc/SMLNJ-C/smlnj-c.ps`. Retrieved October 22, 2010.
19. Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27–30, 2002, Proceedings*, volume 2392 of *LNCS*, pages 134–138. Springer, 2002.
20. Clément Hurlin, Amine Chaieb, Pascal Fontaine, Stephan Merz, and Tjark Weber. Practical proof reconstruction for first-order logic and set-theoretical constructions. In L. Dixon and M. Johansson, editors, *Isabelle Workshop 2007, Proceedings*, pages 2–13, 2007.
21. Daniel Kroening and Ofer Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer, 2008.
22. Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In A. Armando and A. Cimatti, editors, *Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, 2006.
23. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.
24. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised*. MIT Press, May 1997.
25. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – an application of decompilation into logic. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer-Aided Design, 8th International Conference, FMCAD 2008, Proceedings*. IEEE, 2008.
26. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
27. Michael Norrish. Complete integer decision procedures as derived rules in HOL. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Proceedings*, volume 2758 of *LNCS*, pages 71–86. Springer, 2003.
28. Michael Norrish and Konrad Slind. *The HOL System Description*, 2007. Available from `http://hol.sourceforge.net/documentation.html`. Retrieved October 22, 2010.
29. Sam Owre. PVS specification and verification system, July 2008. Available from `http://pvs.csl.sri.com/`. Retrieved October 22, 2010.
30. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction – CADE-11, 11th International Conference on Automated Deduction, Proceedings*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.
31. Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Proceedings*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.
32. Silvio Ranise and Cesare Tinelli. The SMT-LIB standard: Version 1.2, August 2006. Available from `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`. Retrieved October 22, 2010.
33. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Proceedings*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
34. Geoff Sutcliffe and Christian Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
35. Tjark Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, April 2008.
36. Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7:26–40, 2009.
37. Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction, Proceedings*, volume 2392 of *LNCS*, pages 295–313. Springer, 2002.