Bounded Model Generation for Isabelle/HOL

Tjark Weber [1,2]

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany

Abstract

A translation from higher-order logic (on top of the simply typed λ-calculus) to propositional logic is presented, such that the resulting propositional formula is satisfiable iff the HOL formula has a model of a given finite size. A standard SAT solver can then be used to search for a satisfying assignment, and such an assignment can be transformed back into a model for the HOL formula. The algorithm has been implemented in the interactive theorem prover Isabelle/HOL, where it is used to automatically generate countermodels for non-theorems.

Key words: Higher-Order Logic, Finite Model Generation, Interactive Theorem Proving

1 Introduction

Interactive theorem provers have been enhanced with numerous automatic proof procedures for different application domains. However, when an automatic proof attempt fails, the user usually gets little information about the reasons. It may be that an additional lemma needs to be proved, that an induction hypothesis needs to be generalized, or that the formula one is trying to prove is not valid. In such cases an automatic tool that can refute non-theorems would be useful.

This paper presents a translation from higher-order logic to propositional logic (quantifier-free Boolean formulae) such that the propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size, i.e. involving no more than a given number of elements. A standard SAT solver can then be used to search for a satisfying assignment, and if such an assignment is found, it can easily be transformed back into a model for the HOL formula.

[1] This work was supported by the PhD program Logic in Computer Science of the German Research Foundation.

[2] Email: webertj@in.tum.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

An algorithm that uses this translation to generate (counter-)models for HOL formulae has been implemented in the interactive theorem prover Isabelle/HOL [15]. This algorithm is not a (semi-)decision procedure: if a formula does not have a model of a given size, it may still have larger or infinite models. The algorithm's applicability is also limited by its complexity, which is non-elementary for higher-order logic. Nevertheless, formulae that occur in practice often have small models, and the usefulness of an approach similar to the one described in this paper has been confirmed in [10].

Section 2 introduces the syntax and semantics of the logic considered in this paper, a version of higher-order logic on top of the simply typed λ-calculus. The model generation algorithm, and in particular the translation into propositional logic are described in Section 3. We conclude with some final remarks in Section 4.

2 The HOL Logic

Our translation can handle a large fragment of the logic that is underlying the HOL [9] and Isabelle/HOL theorem provers. The logic is originally based on Church's simple theory of types [3]. In this section we present the syntax and set-theoretic semantics of the relevant fragment. A complete account of the HOL logic, including a proof system, can be found in [8].

We distinguish types and terms, intended to denote certain sets and elements of sets respectively. Types σ are given by the following grammar, where α ranges over a countably infinite set TV of type variables:

$$\sigma ::= o \mid \alpha \mid \sigma \to \sigma.$$

Type variables stand for arbitrary non-empty sets. The type o denotes a distinguished two-element set $\{\top, \bot\}$. If σ_1 and σ_2 are types, then $\sigma_1 \to \sigma_2$ is the function type with domain σ_1 and range σ_2. It denotes the set of all total functions from the set denoted by its domain to the set denoted by its range. As usual, $\to$ associates to the right, i.e. $\sigma_1 \to \sigma_2 \to \sigma_3$ is short for $\sigma_1 \to (\sigma_2 \to \sigma_3)$.

We assume a countably infinite set V of variables. A term t_σ of type σ is either an (explicitly typed) variable, logical constant, application, or λ-abstraction. Hence terms are given by the following grammar:

$$t_\sigma ::= x_\sigma \mid c_\sigma \mid (t_{\sigma' \to \sigma}\, t_{\sigma'})_\sigma \mid (\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2},$$

where x_σ ranges over variables, and c_σ is either $\Longrightarrow_{o \to o \to o}$ (implication) or $=_{\sigma' \to \sigma' \to o}$ (equality on σ'), usually written in infix notation. Other logical constants, including $\vee_{o \to o \to o}$, $\wedge_{o \to o \to o}$, $\neg_{o \to o}$, and quantifiers of arbitrary order, can be defined as λ-terms [1]. Terms of type o are called *formulae*.

We now define the semantics of terms. Let t_σ be a term of type σ, and let $tv(t_\sigma) \subseteq TV$ be the set of all type variables that occur in t_σ. tv can be

defined inductively with the help of an auxiliary function tv' that collects the type variables occuring in a type:

$$\begin{aligned} tv'(o) &= \emptyset, \\ tv'(\alpha) &= \{\alpha\}, \\ tv'(\sigma_1 \to \sigma_2) &= tv'(\sigma_1) \cup tv'(\sigma_2), \end{aligned}$$

and

$$\begin{aligned} tv(x_\sigma) &= tv'(\sigma), \\ tv(c_\sigma) &= tv'(\sigma), \\ tv((t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma) &= tv(t_{\sigma' \to \sigma}) \cup tv(t'_{\sigma'}), \\ tv((\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}) &= tv'(\sigma_1) \cup tv(t_{\sigma_2}). \end{aligned}$$

Note that $tv(t_\sigma)$ is not necessarily contained in $tv'(\sigma)$. Types σ with $tv'(\sigma) \neq \emptyset$ and terms t_σ with $tv(t_\sigma) \neq \emptyset$ are called *polymorphic*.

Furthermore, let $fv(t_\sigma) \subseteq V$ be the set of all free variables that occur in t_σ, defined as usual:

$$\begin{aligned} fv(x_\sigma) &= \{x_\sigma\}, \\ fv(c_\sigma) &= \emptyset, \\ fv((t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma) &= fv(t_{\sigma' \to \sigma}) \cup fv(t'_{\sigma'}), \\ fv((\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2}) &= fv(t_{\sigma_2}) \setminus \{x_{\sigma_1}\}. \end{aligned}$$

It is obvious that $tv(t_\sigma)$ and $fv(t_\sigma)$ are finite.

An environment D for t_σ is a function that assigns to each type variable $\alpha \in tv(t_\sigma)$ a non-empty set $D(\alpha)$. The semantics of types w.r.t. this environment is formally given by

$$\begin{aligned} [\![o]\!]_D &= \{\top, \bot\}, \\ [\![\alpha]\!]_D &= D(\alpha), \\ [\![\sigma_1 \to \sigma_2]\!]_D &= [\![\sigma_2]\!]_D^{[\![\sigma_1]\!]_D}. \end{aligned}$$

A variable assignment A for t_σ w.r.t. an environment D maps each variable $x_{\sigma'} \in fv(t_\sigma)$ to an element $A(x_{\sigma'})$ of the set denoted by the type σ'. Given a variable assignment A, a variable $x_{\sigma'} \in V$, and an element d of the set denoted by σ', let $A[x_{\sigma'} \mapsto d]$ be the assignment that maps $x_{\sigma'}$ to d, and $v \neq x_{\sigma'}$ to $A(v)$. Now the semantics of terms w.r.t. an environment D and an assignment A is given by

$$[\![x_\sigma]\!]_D^A = A(x_\sigma),$$

$$[\![\Longrightarrow_{o\to o\to o}]\!]^A_D \text{ is the function that sends } \begin{cases} \top, \top \text{ to } \top \\ \top, \bot \text{ to } \bot \\ \bot, \top \text{ to } \top \\ \bot, \bot \text{ to } \top \end{cases},$$

$$[\![=_{\sigma'\to\sigma'\to o}]\!]^A_D \text{ is the function that sends } x, y \in [\![\sigma']\!]_D$$
$$\text{to } \begin{cases} \top \text{ if } x = y \\ \bot \text{ otherwise} \end{cases},$$

$$[\![(t_{\sigma'\to\sigma}\, t'_{\sigma'})_\sigma]\!]^A_D = [\![t_{\sigma'\to\sigma}]\!]^A_D([\![t'_{\sigma'}]\!]^A_D) \text{ (function application)},$$

$$[\![(\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1\to\sigma_2}]\!]^A_D \text{ is the function that sends each } d \in [\![\sigma_1]\!]_D$$
$$\text{to } [\![t_{\sigma_2}]\!]^{A[x_{\sigma_1}\mapsto d]}_D.$$

Hence the semantics of a term t_σ is an element of the set denoted by the type σ, i.e. $[\![t_\sigma]\!]^A_D \in [\![\sigma]\!]_D$.

3 Bounded Model Generation

The model generation for a HOL formula $\phi = t_o$ proceeds in several steps. We first fix the size of the model by choosing an environment D for ϕ that contains only finite sets. Note that environments are determined uniquely up to isomorphism by the *size* of the sets that they assign to type variables; the *names* of elements are irrelevant. With a fixed finite size for every set denoted by a type variable, *every* type then denotes a finite set: clearly $|o| = 2$, and $|\sigma_1 \to \sigma_2| = |\sigma_2|^{|\sigma_1|}$. Our task now is to find a variable assignment A with $[\![\phi]\!]^A_D = \top$. (To generate a countermodel, we can either consider $\neg\phi$, or – equivalently – search for a variable assignment A with $[\![\phi]\!]^A_D = \bot$.) At this point one can already view bounded model generation as a generalization of satisfiability checking, where the search tree is not necessarily binary, but still finite.

3.1 Translation into Propositional Logic

The input formula ϕ is translated into a propositional formula that is satisfiable if and only if such a variable assignment exists. Propositional formulae are given by the following grammar:

$$\varphi ::= \text{True} \mid \text{False} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi,$$

where p ranges over a countably infinite set of Boolean variables. The translation is by induction over terms. Although our aim is to translate a term of type o into a single propositional formula, a more complex intermediate data structure is needed to translate subterms (which may be of arbitrary type):

we use trees whose leafs are labelled with propositional formulae. The construction of these trees is described in detail in the remainder of this section. A tree of height 1 and width m corresponds to a term whose type is a type variable (denoting a set of size m) or o (for $m = 2$), while an n-ary function or predicate is given by a tree of height $n + 1$. We will show how application and λ-abstraction can be "lifted" from the term level to this intermediate data structure.

To define the translation more precisely, several auxiliary functions are needed. The create function is invoked once for each free variable in ϕ, and returns a tree whose leafs are labelled with Boolean variables. (p) is a placeholder for a fresh Boolean variable, i.e. different occurrences of (p) are replaced by different Boolean variables. The height and width of the tree only depend on the type of the free variable in ϕ. Multiple occurrences of a free variable in ϕ are replaced by identical trees.

$$\begin{aligned}\mathrm{create}(o) &= [(p), (p)],\\ \mathrm{create}(\alpha) &= [(p), \ldots, (p)] \text{ of length } |\alpha|,\\ \mathrm{create}(\sigma_1 \to \sigma_2) &= [\mathrm{create}(\sigma_2), \ldots, \mathrm{create}(\sigma_2)] \text{ of length } |\sigma_1|.\end{aligned}$$

As can be seen from these rules, we use Boolean variables in a unary, rather than in a binary fashion. This means that we need n variables to represent an element of a type of size n, rather than $\lceil \log_2 n \rceil$ variables. However, exactly one of these variables must later be set to True (which keeps the search space for the SAT solver small due to unit propagation [21]), and our encoding allows for a relatively simple translation of application. To ensure that exactly one of the Boolean variables $p_1, \ldots, p_n$ is set to True, a propositional formula

$$\mathrm{wf}_{[p_1,\ldots,p_n]} = \left(\bigvee_{i=1}^{n} p_i\right) \wedge \bigwedge_{\substack{i,j=1\\ i\neq j}}^{n} (\neg p_i \vee \neg p_j)$$

is constructed for each tree of the form $[p_1, \ldots, p_n]$ that is returned by a call to create(o) or create(α). This formula is later conjoined with the result of the translation.

TT and FF are trees corresponding to $\top$ and $\bot$, respectively.

$$\begin{aligned}\mathrm{TT} &= [\mathrm{True}, \mathrm{False}],\\ \mathrm{FF} &= [\mathrm{False}, \mathrm{True}].\end{aligned}$$

The k-th unit vector of length n with entries True and False is given by uv_k^n, and likewise UV_k^n is the k-th unit vector of length n with entries TT and FF.

$$\delta_k^n = \begin{cases} \text{True if } n = k \\ \text{False otherwise} \end{cases},$$
$$\mathrm{uv}_k^n = [\delta_1^k, \ldots, \delta_n^k],$$
$$\Delta_k^n = \begin{cases} \text{TT if } n = k \\ \text{FF otherwise} \end{cases},$$
$$\mathrm{UV}_k^n = [\Delta_1^k, \ldots, \Delta_n^k].$$

These unit vectors are used to build trees whose leafs are labelled with propositional constants (True/False) only, representing specific (i.e. the first, second, ...) elements of the domains under consideration. Also note that $\mathrm{TT} = \mathrm{uv}_1^2$, and $\mathrm{FF} = \mathrm{uv}_2^2$.

consts(σ) returns a list of length $|\sigma|$, containing one representing tree for every element in $[\![\sigma]\!]_D$. pick($[x_1, \ldots, x_n]$) – where each x_i is again a list – is an auxiliary function that returns a list containing all possible choices of one element from each list x_i. For the special case $x_1 = \ldots = x_n$, this corresponds to all functions from an n-element set to elements of x_1.

$$\begin{aligned} \mathrm{consts}(o) &= [\mathrm{TT}, \mathrm{FF}], \\ \mathrm{consts}(\alpha) &= [\mathrm{uv}_1^{|\alpha|}, \ldots, \mathrm{uv}_{|\alpha|}^{|\alpha|}], \\ \mathrm{consts}(\sigma_1 \to \sigma_2) &= \mathrm{pick}([\underbrace{\mathrm{consts}(\sigma_2), \ldots, \mathrm{consts}(\sigma_2)}_{|\sigma_1|}]). \end{aligned}$$

The functions described so far are sufficient to define the translation for terms without application. The translation of application, however, requires further helper functions. all returns the conjunction of a list of propositional formulae. map(f, l) applies the unary function f to every element in a list l. Likewise, treemap(f, t) applies f to every leaf in a tree t. merge(g, t_1, t_2) merges two trees t_1 and t_2 by applying a binary function g to corresponding leafs in t_1 and t_2. Both trees must have the same "structure", i.e. differ at most in the formulae that they contain (but not in their height or width).

$$\begin{aligned} \mathrm{all}([\varphi_1, \ldots, \varphi_n]) &= \varphi_1 \wedge \ldots \wedge \varphi_n, \\ \mathrm{map}(f, [\varphi_1, \ldots, \varphi_n]) &= [f(\varphi_1), \ldots, f(\varphi_n)], \\ \mathrm{treemap}(f, [\varphi_1, \ldots, \varphi_n]) &= [f(\varphi_1), \ldots, f(\varphi_n)], \\ \mathrm{treemap}(f, [t_1, \ldots, t_n]) &= [\mathrm{treemap}(f, t_1), \ldots, \mathrm{treemap}(f, t_n)], \\ \mathrm{merge}(g, [\varphi_1^1, \ldots, \varphi_n^1], [\varphi_1^2, \ldots, \varphi_n^2]) &= [g(\varphi_1^1, \varphi_1^2), \ldots, g(\varphi_n^1, \varphi_n^2)], \\ \mathrm{merge}(g, [t_1^1, \ldots, t_n^1], [t_1^2, \ldots, t_n^2]) &= [\mathrm{merge}(g, t_1^1, t_1^2), \ldots, \mathrm{merge}(g, t_n^1, t_n^2)]. \end{aligned}$$

enum(t), for t a tree representing an element of $[\![\sigma]\!]_D$, computes a list of propositional formulae $[\varphi_1, \ldots, \varphi_{|\sigma|}]$ expressing that t represents the first, ..., $|\sigma|$-th element of $[\![\sigma]\!]_D$.

$$\begin{aligned} \mathrm{enum}([\varphi_1, \ldots, \varphi_n]) &= [\varphi_1, \ldots, \varphi_n], \\ \mathrm{enum}([t_1, \ldots, t_n]) &= \mathrm{map}(\mathrm{all}, \mathrm{pick}([\mathrm{enum}(t_1), \ldots, \mathrm{enum}(t_n)])). \end{aligned}$$

Functions are represented by trees of height greater than 1. Intuitively, when a function is applied to the i-th element of its domain, the result is given by the i-th subtree of the tree representing the function. $\mathrm{apply}(t, [\varphi_1, \ldots, \varphi_n])$, where t is a tree representing a function, and φ_i is true iff the function's argument is equal to the i-th element of the function's domain, builds a tree whose leafs are labelled with propositional formulae that simulate the selection of the correct subtree of t.

$$\begin{aligned}\mathrm{apply}([t_1], [\varphi_1]) &= \mathrm{treemap}((\underline{\lambda}\varphi.\, \varphi \wedge \varphi_1), t_1),\\ \mathrm{apply}([t_1, t_2, \ldots, t_n], [\varphi_1, \varphi_2, \ldots, \varphi_n]) &= \mathrm{merge}((\underline{\lambda}\varphi\, \varphi'.\, \varphi \vee \varphi'),\\ &\qquad \mathrm{apply}([t_1], [\varphi_1]), \mathrm{apply}([t_2, \ldots, t_n], [\varphi_2, \ldots, \varphi_n])).\end{aligned}$$

Above $\underline{\lambda}$ is a meta-symbol used to denote a function (as opposed to the λ that is part of the term syntax), while $\vee$ and $\wedge$ are constructors of propositional formulae.

We are now ready to define the translation $\mathcal{T}_D$ from terms to trees of propositional formulae. The translation is parameterized by a partial assignment B of trees to bound variables. Initially this partial assignment is empty, and it is extended whenever the translation descends into the body of a λ-abstraction. The translation is given by the following rules:

$$\mathcal{T}_D^B(x_\sigma) = \begin{cases} B(x_\sigma) & \text{if } x_\sigma \in \mathrm{dom}\, B \\ \mathrm{create}(\sigma) & \text{otherwise} \end{cases},$$

$$\begin{aligned}\mathcal{T}_D^B(\Longrightarrow_{o\to o\to o}) &= [[\mathrm{TT}, \mathrm{FF}], [\mathrm{TT}, \mathrm{TT}]],\\ \mathcal{T}_D^B(=_{\sigma'\to\sigma'\to o}) &= [\mathrm{UV}_1^{|\sigma'|}, \ldots, \mathrm{UV}_{|\sigma'|}^{|\sigma'|}],\\ \mathcal{T}_D^B((t_{\sigma'\to\sigma}\, t'_{\sigma'})_\sigma) &= \mathrm{apply}(\mathcal{T}_D^B(t_{\sigma'\to\sigma}), \mathrm{enum}(\mathcal{T}_D^B(t'_{\sigma'}))),\\ \mathcal{T}_D^B((\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1\to\sigma_2}) &= [\mathcal{T}_D^{B[x_{\sigma_1}\mapsto d_1]}(t_{\sigma_2}), \ldots, \mathcal{T}_D^{B[x_{\sigma_1}\mapsto d_{|\sigma_1|}]}(t_{\sigma_2})],\\ &\qquad \text{where } [d_1, \ldots, d_{|\sigma_1|}] = \mathrm{consts}(\sigma_1).\end{aligned}$$

Since ϕ is a term of type o, the result $\mathcal{T}_D^\emptyset(\phi)$ of the translation must be a tree of the form $[\mathcal{T}_D^\emptyset(\phi)^\top, \mathcal{T}_D^\emptyset(\phi)^\bot]$ for some propositional formulae $\mathcal{T}_D^\emptyset(\phi)^\top, \mathcal{T}_D^\emptyset(\phi)^\bot$.

Proposition 3.1 *Soundness, Completeness. Let* $* \in \{\top, \bot\}$*, and let* WF *be the conjunction of all* wf*-formulae constructed during the translation. Then* $[\![\phi]\!]_D^A = *$ *for some variable assignment* A *if and only if* $\mathrm{WF} \wedge \mathcal{T}_D^\emptyset(\phi)^*$ *is satisfiable.*

The theorem can be proved by generalization from formulae to terms of arbitrary type, followed by induction over the term. We omit the details.

3.2 Finding a Satisfying Assignment

Satisfiability can be tested with an off-the-shelf SAT solver. To this end translations into DIMACS SAT and DIMACS CNF format [6] have been implemented. The translation into SAT format is trivial, whereas CNF format

(supported by zChaff [14], BerkMin [7] and other state-of-the-art solvers) requires the Boolean formula to be in conjunctive normal form. We translate into definitional CNF [19] to avoid an exponential blowup at this stage, introducing auxiliary Boolean variables where necessary. A more sophisticated CNF conversion might further enhance the performance of our approach [11].

Isabelle/HOL runs on a number of different platforms, and installation should be as simple as possible. Therefore we have also implemented a naive DPLL-based [5,21] SAT solver in Isabelle. This solver is not meant to replace the external solver for serious applications, but it has proved to be efficient enough for small examples. Hence it allows users to experiment with the countermodel generation without them having to worry about the installation of an additional tool.

If the SAT solver cannot find a satisfying assignment, the translation is repeated for a larger environment. The user can specify several termination conditions: a maximal size for sets in the environment, a limit on the number of Boolean variables to be used, a runtime limit. The order in which we enumerate environments guarantees that if the SAT solver is complete, a model will be found that is minimal w.r.t. the total size of its domains. Of course this is not necessarily true for incomplete (e.g. stochastic) SAT solvers.

3.3 Example Translation

Consider the formula $\phi = ((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha} =_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to o} y_{\alpha\to\alpha})_o$. Its only type variable is α, and its only free variable is $y_{\alpha\to\alpha}$. In an environment D with $|\alpha| = 2$ (and hence $|\alpha\to\alpha| = 2^2 = 4$), the subterms of ϕ are translated into the following trees:

$$\begin{aligned}\mathcal{T}^{\emptyset}_D((\lambda x_\alpha.\, x_\alpha)_{\alpha\to\alpha}) &= [[\text{True}, \text{False}], [\text{False}, \text{True}]],\\ \mathcal{T}^{\emptyset}_D(=_{(\alpha\to\alpha)\to(\alpha\to\alpha)\to o}) &= [\text{UV}^4_1, \text{UV}^4_2, \text{UV}^4_3, \text{UV}^4_4],\\ \mathcal{T}^{\emptyset}_D(y_{\alpha\to\alpha}) &= [[y_0, y_1], [y_2, y_3]]\end{aligned}$$

with four Boolean variables y_0, y_1, y_2, y_3. Additionally two wf-formulae are constructed, namely

$$\text{wf}_{[y_0,y_1]} = (y_0 \vee y_1) \wedge (\neg y_0 \vee \neg y_1)$$

and

$$\text{wf}_{[y_2,y_3]} = (y_2 \vee y_3) \wedge (\neg y_2 \vee \neg y_3).$$

Using the translation rule for application, we then obtain (Boolean formulae equivalent to)

$$\mathcal{T}^{\emptyset}_D(\phi)^{\top} = y_0 \wedge y_3$$

and

$$\mathcal{T}^{\emptyset}_D(\phi)^{\bot} = (y_0 \wedge y_2) \vee (y_1 \wedge y_2) \vee (y_1 \wedge y_3).$$

Hence a possible satisfying assignment for $\mathcal{T}^{\emptyset}_D(\phi)^{\bot} \wedge \text{wf}_{[y_0,y_1]} \wedge \text{wf}_{[y_2,y_3]}$ is given by $\{y_0 \mapsto \text{True}, y_1 \mapsto \text{False}, y_2 \mapsto \text{True}, y_3 \mapsto \text{False}\}$. Assuming $[\![\alpha]\!]_D = \{a_0, a_1\}$,

this assignment corresponds to an interpretation of $y_{\alpha\to\alpha}$ as the function that maps both a_0 and a_1 to a_0.

3.4 Some Extensions: Sets, Hilbert's Choice, and Datatypes

Several extensions to the logic described in Section 2 can straightforwardly be integrated into our framework. The type σ set of sets with elements from σ is isomorphic to $\sigma \to o$. Set membership $x \in P$ becomes predicate application $P\,x$, and set comprehension $\{x.\,P\}$ can be translated simply as P.

Hilbert's choice operator, ϵ, is a polymorphic constant of type $(\sigma \to o) \to \sigma$, satisfying the axiom

$$\phi_\epsilon: \quad (\exists x.\,P\,x) \implies P(\epsilon P).$$

Similarly, *The*, also a constant of type $(\sigma \to o) \to \sigma$, satisfies

$$\phi_{The}: \quad (\mathit{The}\,x.\,x = a) = a,$$

and *arbitrary* is a completely unspecified polymorphic constant. For the purpose of our translation $\mathcal{T}_D$, we can treat these logical constants just like free variables, and introduce Boolean variables that determine their interpretation. For ϵ and *The*, we then translate the conjunction of the original formula ϕ with the relevant axiom (i.e. $\phi_\epsilon \wedge \phi$ or $\phi_{The} \wedge \phi$, respectively, or $\phi_\epsilon \wedge \phi_{The} \wedge \phi$ if both ϵ and *The* occur in ϕ). Type variables in ϕ_ϵ (or ϕ_{The}) are instantiated to match the type of ϵ (or *The*) in ϕ.

Isabelle/HOL allows the definition of inductive datatypes [2]. In general, inductive datatypes with free constructors require an infinite model. We treat these datatypes by only considering a finite fragment (e.g. natural numbers up to an upper bound, or lists up to a certain length). A detailed description of the translation of inductive datatypes will be given elsewhere. We remark that if the inductive datatype occurs only positively in the input formula, then a model found for a fragment can always be extended to an infinite model for the full datatype.

However, many important datatypes are non-recursive, and for these, the situation is simpler. Examples are the type σ option, which augments a given type σ by a new element, product types $\sigma_1 \times \sigma_2$, and sum types $\sigma_1 + \sigma_2$. The general syntax of a non-recursive datatype definition is given by

$$(\alpha_1, \ldots, \alpha_n)\sigma ::= C_1\,\sigma_1^1 \ldots \sigma_{m_1}^1 \mid \ldots \mid C_k\,\sigma_1^k \ldots \sigma_{m_k}^k,$$

where the C_i are the datatype's constructors, the σ_j^i specify their argument types, and all σ_j^i only refer to previously defined types and type variables from $\alpha_1, \ldots, \alpha_n$. Such a datatype can be interpreted in a finite model; its size is equal to $S := \sum_{i=1}^{k} \prod_{j=1}^{m_i} |\sigma_j^i|$. Hence an element of this datatype can be represented by a tree of height 1 and width S, and a datatype constructor C_i

is a function of type $\sigma_1^i \to \ldots \to \sigma_{m_i}^i \to (\alpha_1, \ldots, \alpha_n)\sigma$, representable by a tree of height $m_i + 1$.

3.5 Some Optimizations

We briefly describe some optimizations in the implementation of the translation $\mathcal{T}_D$. None of them affect soundness or completeness of the algorithm.

The Boolean formulae that are constructed during the translation process are simplified on the fly, using basic algebraic laws of $\neg$, $\vee$, $\wedge$, True and False. Closed HOL formulae simply become True or False. The SAT solver is used only to search for an interpretation of *free* variables.

Variables of a type with size 1 can be represented by [True], using no Boolean variable at all (instead of one Boolean variable x together with a $\mathrm{wf}_{[x]}$-formula x). While this has little effect at the SAT solver level due to unit propagation, it allows a more extensive simplification of the constructed Boolean formulae. Similarly variables of a type with size 2, including variables of type o, can be represented by a tree of the form $[x, \neg x]$, rather than by a tree $[x_0, x_1]$ and a $\mathrm{wf}_{[x_0,x_1]}$-formula $(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1)$.

More importantly, we avoid unfolding the definition of logical constants (i.e. True_o, False_o, $\neg_{o\to o}$, $\wedge_{o\to o\to o}$, $\vee_{o\to o\to o}$, $\forall_{(\sigma\to o)\to o}$, $\exists_{(\sigma\to o)\to o}$) as λ-terms as far as possible. Instead these constants are replaced directly by their counterparts in propositional logic. Since every type is finite, quantifiers of arbitrary order can be replaced by a finite conjunction or disjunction.

The latter leads to a more general optimization technique, applicable also to other functions and predicates (including e.g. equality): namely specialization of the rule for function application to particular functions. While any given function can be represented by a tree, it is often more efficient to implement a particular function's action on its arguments, assuming these arguments are given as trees already, than to use the general translation rule and apply it to the tree representing the function. For $=_{\sigma\to\sigma\to o}$ this avoids creating a tree whose size is proportional to $|\sigma|^2$, and instead uses a function that operates on trees representing elements of $[\![\sigma]\!]_D$, their size proportional to $|\sigma|$ (or possibly to $\log|\sigma|$) only.

3.6 Examples

Table 1 shows some examples of formulae for which our algorithm can automatically find a countermodel. Type annotations are suppressed, and functions in the countermodel are given by their graphs. "$\exists!$" denotes unique existence, defined as usual:

$$\exists!x.\, P\, x \;\equiv\; \exists x.\, P\, x \wedge (\forall y.\, P\, y \implies y = x).$$

The countermodels are rather small, and were all found within a few milliseconds. The main purpose of these examples is to illustrate the expressive

Property/Formula	Countermodel
"Every function that is onto is invertible." $(\forall y.\, \exists x.\, f\, x = y) \implies (\exists g.\, \forall x.\, g\, (f\, x) = x)$	$D(\alpha) = \{a_0, a_1\}, D(\beta) = \{b_0\}$ $f = \{(a_0, b_0), (a_1, b_0)\}$
"There exists a unique choice function." $(\forall x.\, \exists y.\, P\, x\, y) \implies (\exists! f.\, \forall x.\, P\, x\, (f\, x))$	$D(\alpha) = \{a_0\}, D(\beta) = \{b_0, b_1\}$ $P = \{(a_0, \{(b_0, \text{True}), (b_1, \text{True})\})\}$
"The transitive closure of $A \cap B$ is equal to the intersection of the transitive closures of A and B."	$D(\alpha) = \{a_0, a_1\}$ $A = \{(a_0, a_1), (a_1, a_0), (a_1, a_1)\}$ $B = \{(a_0, a_0), (a_1, a_0), (a_1, a_1)\}$

Table 1
Examples

power of the underlying logic.

4 Conclusions and Future Work

We have presented a translation from higher-order logic to propositional formulae, such that the resulting propositional formula is satisfiable if and only if the HOL formula has a model of a given finite size. A working implementation of this translation, consisting of roughly 2,800 lines of code written in Standard ML [13], is available in the Isabelle/HOL theorem prover. A standard SAT solver can be used to search for a satisfying assignment for the propositional formula, and if such an assignment is found, it can be transformed into a model for the HOL formula. This allows for the automatic generation of finite countermodels for non-theorems in Isabelle/HOL. A similar translation has been discussed before [10]; the main contributions of this paper are its extension to higher-order logic and the seamless integration with a popular interactive theorem prover.

So far we have applied the technique only to relatively small examples. The applicability of the algorithm is limited by its non-elementary complexity. We believe that the algorithm can still be useful for practical purposes, since many formulae have small models. To substantiate this claim, and to further evaluate the performance of our approach, we plan to carry out some larger case studies, possibly from the area of cryptographic protocol verification [16,17].

We also plan to incorporate further optimizations [4,18], and to extend the translation to other Isabelle/HOL constructs: most notably the full language of HOL, including type operators [8], but also axiomatic type classes [20], inductively defined sets, and recursive functions.

An orthogonal approach that would be interesting to evaluate, both in terms of performance and feasibility, is the use of an external (first-order) model generator. The necessary translation from HOL to first-order logic

could be based on recent work by Meng and Paulson [12].

Acknowledgments The author would like to thank Martin Strecker, Tobias Nipkow and the anonymous referees for their valuable comments.

References

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, second edition, July 2002.

[2] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1999.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[6] DIMACS satisfiability suggested format, 1993. Available from ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc.

[7] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[9] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems Series*, pages 49–70. Elsevier, 1994.

[10] Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, pages 130–139, San Diego, November 2000.

[11] Paul Jackson and Daniel Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2004, APES Research Group, March 2004.

[12] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michael Rusinowitch, editors, *Automated Reasoning – Second International Joint Conference, IJCAR 2004*,

volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 372–384. Springer, 2004.

[13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Design Automation Conference*, Las Vegas, June 2001.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[17] Graham Steel, Alan Bundy, and Ewen Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2), 2002.

[18] Tanel Tammet. Finite model building: improvements and comparisons. In *CADE-19, Workshop W4, Model Computation – Principles, Algorithms, Applications*, 2003.

[19] G. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1970.

[20] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

[21] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *Lecture Notes in Computer Science*. Springer, 2002.