Finite Model Generation for Isabelle/HOL

Using a SAT Solver

Tjark Weber

webertj@in.tum.de

Technische Universität München

Winterhütte, März 2004

Isabelle

Isabelle is a generic proof assistant:

- Highly flexible
- Interactive
- Automatic proof procedures
- Advanced user interface
- Readable proofs
- Large theories of formal mathematics

Finite Model Generation

Theorem proving: from formulae to proofs

Finite model generation: *from formulae to models*

Applications:

- Showing the consistency of a specification
- *Finding counterexamples to false conjectures*
- Solving open mathematical problems
- Guiding resolution-based provers

Isabelle/HOL

HOL: higher-order logic on top of polymorphic simply-typed λ-calculus

Isabelle/HOL

HOL: higher-order logic on top of polymorphic simply-typed λ-calculus

Simply-typed λ-calculus:

- Types: $\tau ::= \mathbb{B} \mid \alpha \mid \beta \mid \ldots \mid \tau \Rightarrow \tau$
- Terms: $\Lambda ::= x \mid y \mid \ldots \mid \lambda x.\Lambda \mid (\Lambda\,\Lambda)$
- Typing rules: $\dfrac{x{:}\tau_1 \vdash \Lambda{:}\tau_2}{\lambda x.\Lambda{:}\tau_1 \Rightarrow \tau_2}$ $\qquad$ $\dfrac{\Lambda_1{:}\tau_1 \Rightarrow \tau_2 \qquad \Lambda_2{:}\tau_1}{(\Lambda_1\,\Lambda_2){:}\tau_2}$

Isabelle/HOL

HOL: higher-order logic on top of polymorphic simply-typed λ-calculus

Simply-typed λ-calculus:

- Types: $\tau ::= \mathbb{B} \mid \alpha \mid \beta \mid \ldots \mid \tau \Rightarrow \tau$
- Terms: $\Lambda ::= x \mid y \mid \ldots \mid \lambda x.\Lambda \mid (\Lambda\,\Lambda)$
- Typing rules: $\dfrac{x{:}\tau_1 \vdash \Lambda{:}\tau_2}{\lambda x.\Lambda{:}\tau_1 \Rightarrow \tau_2}$ $\quad \dfrac{\Lambda_1{:}\tau_1 \Rightarrow \tau_2 \qquad \Lambda_2{:}\tau_1}{(\Lambda_1\,\Lambda_2){:}\tau_2}$

The logical constants

$$\mathtt{True} \mid \mathtt{False} \mid \neg \mid \wedge \mid \vee \mid \rightarrow \mid = \mid \forall \mid \exists \mid \exists!$$

are definable.

The Semantics of HOL

A (finite) model for a HOL formula is given by

- (finite) *sets of (first-order) individuals*, and
- an *interpretation* of the formula's variables.

Finite model generation is a generalization of satisfiability checking, where the search tree is not necessarily binary (as in the case of SAT).

Overview

Input: HOL formula ϕ

Output: either a model for ϕ, or “no model found”

Overview

Input: HOL formula ϕ

1. Fix the size of the model.
2. Translate ϕ into a boolean formula that is satisfiable iff ϕ has a model of the given size.
3. Use a SAT solver to search for a satisfying assignment.
4. If no assignment was found, increase the size of the model and repeat.

Output: either a model for ϕ, or “no model found”

1. Fixing the Size of the Model

Fix a positive integer for every type variable that occurs in the typing of ϕ.

Every type then has a finite size:

- $|\mathbb{B}| = 2$
- $|\alpha|$, $|\beta|$, $\ldots$ is given by the model
- $|\sigma \Rightarrow \tau| = |\tau|^{|\sigma|}$

2. Translation into a Boolean Formula

Boolean formulae:

$$\varphi ::= \texttt{True} \mid \texttt{False} \mid p \mid q \mid \ldots \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

2. Translation into a Boolean Formula

Boolean formulae:

$$\varphi ::= \texttt{True} \mid \texttt{False} \mid p \mid q \mid \ldots \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

Idea: Translate a HOL term Λ into a *tree of lists of boolean formulae*. The interpretation of the boolean variables in the tree determines the interpretation of Λ.

2. Translation into a Boolean Formula

Boolean formulae:

$$\varphi ::= \texttt{True} \mid \texttt{False} \mid p \mid q \mid \ldots \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

Idea: Translate a HOL term Λ into a *tree of lists of boolean formulae*. The interpretation of the boolean variables in the tree determines the interpretation of Λ.

1. A variable x of type α becomes a list of boolean variables $[x_1, \ldots, x_{|\alpha|}]$ of length $|\alpha|$.

   Idea: x_i is true iff x is to be interpreted as the i-th element of α.

   Add clauses to make sure that exactly one variable x_i $(1 \leq i \leq |\alpha|)$ is true.

2. Translation into a Boolean Formula

2. A variable of type $\sigma \Rightarrow \tau$ becomes a tree whose root has $|\sigma|$ children, each one being a (fresh) tree for a variable of type τ.

2. Translation into a Boolean Formula

2. A variable of type $\sigma \Rightarrow \tau$ becomes a tree whose root has $|\sigma|$ children, each one being a (fresh) tree for a variable of type τ.
3. A λ-abstraction $\lambda x.\Lambda$ of type $\sigma \Rightarrow \tau$ becomes a tree whose root has $|\sigma|$ children, each one being a tree for Λ with x bound to a tree for the corresponding (first, second, $\ldots$, $|\sigma|$-th) constant in σ.

2. Translation into a Boolean Formula

4. An application $(S\,T)$ is translated as follows:
   (a) Pick the first formula from every leaf in the tree for T.
   (b) Compute the conjunction of these formulae.
   (c) Compute the “conjunction” with the first child in S.
   (d) Repeat for every child in S (with the *corresponding* choice of formulae from T).
   (e) Compute the “disjunction” of all children.

2. Translation into a Boolean Formula

4. An application $(S\,T)$ is translated as follows:
   (a) Pick the first formula from every leaf in the tree for T.
   (b) Compute the conjunction of these formulae.
   (c) Compute the “conjunction” with the first child in S.
   (d) Repeat for every child in S (with the *corresponding* choice of formulae from T).
   (e) Compute the “disjunction” of all children.

Example: $S :: \alpha \Rightarrow \beta$, $T :: \alpha$, $|\alpha| = 2$, $|\beta| = 3$

$S = [[s_1^1, s_2^1, s_3^1], [s_1^2, s_2^2, s_3^2]]$
$T = [t_1, t_2]$

2. Translation into a Boolean Formula

4. An application $(S\,T)$ is translated as follows:
   (a) Pick the first formula from every leaf in the tree for T.
   (b) Compute the conjunction of these formulae.
   (c) Compute the "conjunction" with the first child in S.
   (d) Repeat for every child in S (with the *corresponding* choice of formulae from T).
   (e) Compute the "disjunction" of all children.

Example: $S :: \alpha \Rightarrow \beta$, $T :: \alpha$, $|\alpha| = 2$, $|\beta| = 3$

$$S = [[s_1^1, s_2^1, s_3^1], [s_1^2, s_2^2, s_3^2]]$$
$$T = [t_1, t_2]$$

$$(S\,T) = [s_1^1 \wedge t_1 \qquad , s_2^1 \wedge t_1 \qquad , s_3^1 \wedge t1 \qquad]$$

2. Translation into a Boolean Formula

4. An application $(S\,T)$ is translated as follows:
   (a) Pick the first formula from every leaf in the tree for T.
   (b) Compute the conjunction of these formulae.
   (c) Compute the "conjunction" with the first child in S.
   (d) Repeat for every child in S (with the *corresponding* choice of formulae from T).
   (e) Compute the "disjunction" of all children.

Example: $S :: \alpha \Rightarrow \beta$, $T :: \alpha$, $|\alpha| = 2$, $|\beta| = 3$

$$S = [[s_1^1, s_2^1, s_3^1], [s_1^2, s_2^2, s_3^2]]$$
$$T = [t_1, t_2]$$

$$(S\,T) = [s_1^1 \wedge t_1 \vee s_1^2 \wedge t_2, s_2^1 \wedge t_1 \vee s_2^2 \wedge t_2, s_3^1 \wedge t_1 \vee s_3^2 \wedge t_2]$$

3. The SAT Solver

Both *internal* and *external* SAT solvers are supported.

3. The SAT Solver

Both *internal* and *external* SAT solvers are supported.

Pros of an internal solver:

- Easy installation
- Compatibility
- Fast enough for simple examples

3. The SAT Solver

Both *internal* and *external* SAT solvers are supported.

Pros of an internal solver:

- Easy installation
- Compatibility
- Fast enough for simple examples

Pros of an external solver:

- Efficiency
- Advances in SAT solver technology are “for free”

The Internal Solver

Based on the *DPLL* procedure (Davis-Putnam-Logemann-Loveland, 1962)

```
dpll(θ:partial assignment, φ:formula) {
   (θ′,φ′) := simplify_and_deduce(θ,φ);
   if φ′=True then return θ′
   else if φ′=False then return UNSATISFIABLE
   else {
     x := pick_fresh_variable(θ′,φ′);
     result := dpll(θ′[x ↦ False], φ′);
     if result=UNSATISFIABLE then
       return dpll(θ′[x ↦ True], φ′)
     else return result
   }
}
```

External Solvers

Interface:

- Input/output: via text files
- Execution: via a system call

Supported input formats:

- DIMACS SAT
- DIMACS CNF

DIMACS SAT

- Arbitrary boolean formulae allowed

```
c Example SAT format file in DIMACS format
c
p sat 4
(*(+( 2 3- (( 4 ) ) )
+( -4 )
+( 2 3 4 ) ))
```

DIMACS CNF

- Formula must be in CNF ($\bigwedge \bigvee (\neg) p$)

```
c Example CNF format file in DIMACS format
c
p cnf 4 3
2 3 -4 0
-4 0
2 3 4 0
```

DIMACS CNF

- Formula must be in CNF ($\bigwedge \bigvee (\neg) p$)

```
c Example CNF format file in DIMACS format
c
p cnf 4 3
2 3 -4 0
-4 0
2 3 4 0
```

Most SAT solvers *only* support CNF format!

Translation into CNF

1. Translate into NNF
   - $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
   - $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
   - $\neg\neg P \equiv P$
2. Translate into CNF
   - $(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$

Translation into CNF

1. Translate into NNF
   - $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
   - $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
   - $\neg\neg P \equiv P$
2. Translate into CNF
   - $(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$

This translation can cause an *exponential blow-up* of the formula.

Translation into CNF

1. Translate into NNF
   - $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
   - $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
   - $\neg\neg P \equiv P$
2. Translate into CNF
   - $(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$

This translation can cause an exponential blow-up of the formula.

Solution: *Definitional CNF*

$$(P \wedge Q) \vee R \stackrel{sat}{\equiv} (P \vee p) \wedge (Q \vee p) \wedge (R \vee \neg p)$$

Some Optimizations

- Hard-coded translation for logical constants
- Only *one* boolean variable is used for variables of type $\mathbb{B}$
- On-the-fly simplification of the boolean formula (e.g. closed HOL formulae simply become `True`/`False`)

A Simple Extension: Sets

Sets are interpreted as characteristic functions.

- $\alpha\ \texttt{set} \cong \alpha \Rightarrow \mathbb{B}$
- $x \in P \cong P\,x$
- $\{x.\,P\,x\} \cong P$

Soundness and Completeness

If the SAT solver is sound/complete, we have ...

- *Soundness*: If the algorithm returns “model found”, the given formula has a finite model.
- *Completeness*: If the given formula has a finite model, the algorithm will find it (given enough time).

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model
- `maxvars`: max. number of boolean variables
- `satsolver`: name of the SAT solver to be used

All parameters can be set globally with `refute_params`.

Future Work

- A better translation:
  - polynomial-time
  - logarithmic number of boolean variables
  - types encoded as terms
- Support for other HOL constructs:
  - axioms
  - typedefs
  - inductive datatypes
  - inductively defined sets
  - recursive functions