Finite Model Generation for Isabelle/HOL

Using a SAT Solver

Tjark Weber

`webertj@in.tum.de`

Technische Universität München

Club2, 16.1.2004

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

The Semantics of HOL

A *finite model* for a HOL formula is given by

- a finite *set of (first-order) individuals* and
- an *interpretation* of the formula's logical constants and 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”

Input: A Fragment of HOL

Simply-typed λ-calculus:

- Types: $\tau ::= \mathbb{B} \mid \alpha \mid \beta \mid \ldots \mid \tau \Rightarrow \tau$
- Terms: $\Lambda ::= x \mid y \mid \ldots \mid \lambda x.\Lambda \mid (\Lambda\,\Lambda)$

Input: A Fragment of HOL

Simply-typed λ-calculus:

- Types: $\tau ::= \mathbb{B} \mid \alpha \mid \beta \mid \ldots \mid \tau \Rightarrow \tau$
- Terms: $\Lambda ::= x \mid y \mid \ldots \mid \lambda x.\Lambda \mid (\Lambda\,\Lambda)$

The logical constants

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

are definable (see file "HOL.thy").

Input: A Fragment of HOL

Simply-typed λ-calculus:

- Types: $\tau ::= \mathbb{B} \mid \alpha \mid \beta \mid \ldots \mid \tau \Rightarrow \tau$
- Terms: $\Lambda ::= x \mid y \mid \ldots \mid \lambda x.\Lambda \mid (\Lambda\,\Lambda)$

The logical constants

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

are definable (see file “HOL.thy”).

Not allowed (yet):

- Other type constructors
- Other constants

1. Fixing the Size of the Model

- A typing may contain several type variables.
- HOL types are assumed to be non-empty (e.g. $(\forall x.\, P\, x) \longrightarrow (\exists x.\, P\, x)$ is a theorem).

Fix a positive integer k. Consider *all possible partitions* of k into n parts, where n is the number of type variables that occur in the typing of ϕ.

1. Fixing the Size of the Model

- A typing may contain several type variables.
- HOL types are assumed to be non-empty (e.g. $(\forall x.\, P\, x) \longrightarrow (\exists x.\, P\, x)$ is a theorem).

Fix a positive integer k. Consider *all possible partitions* of k into n parts, where n is the number of type variables that occur in the typing of ϕ.

Example: $k = 3$, type variables α and β (i.e. $n = 2$)

1. $|\alpha| = 1, |\beta| = 2$
2. $|\alpha| = 2, |\beta| = 1$

1. Fixing the Size of the Model

Every type now 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 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 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 replaced by 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. Using an External SAT Solver

Pros of an external tool:

- Greatly reduces development time
- Advances in SAT solver technology are “for free”

3. Using an External SAT Solver

Pros and cons of an external tool:

- Greatly reduces development time
- Advances in SAT solver technology are “for free”
- Legal issues (copyright)
- Installation
- Compatibility

3. Using an External SAT Solver

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*

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

3. Using an External SAT Solver

Supported output format:

- A line containing a *message of success* (e.g. "`Instance satisfiable`"), followed by
- the *satisfying assignment*, given by a list of integers:
  - `i` means "variable i is true"
  - `-i` means "variable i is false"

3. Using an External SAT Solver

Supported output format:

- A line containing a *message of success* (e.g. "`Instance satisfiable`"), followed by
- the *satisfying assignment*, given by a list of integers:
  - `i` means "variable i is true"
  - `-i` means "variable i is false"

Example:

```
Z-Chaff Version: ZChaff 2003.11.04
Solving sample.cnf ......
3 Clauses are true, Verify Solution successful. Instance satisfiable
1 2 3 -4
Max Decision Level 0
...
```

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`)

Soundness and Completeness

- *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).

Soundness and Completeness

- *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).

Caveat: We have soundness/completeness only if the SAT solver is sound/complete!

A Simple Extension: Sets

Sets are interpreted as characteristic functions.

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

Inductive Datatypes

Problem: IDTs may require an *infinite* model

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\texttt{nat}^i = \texttt{Zero} \mid \texttt{Suc Zero} \mid \ldots \mid \texttt{Suc}^{i-1}\ \texttt{Zero}$

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\texttt{nat}^i = \texttt{Zero} \mid \texttt{Suc Zero} \mid \ldots \mid \texttt{Suc}^{i-1}\ \texttt{Zero}$

- Models may be *spurious*

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\texttt{nat}^i = \texttt{Zero}$ | $\texttt{Suc Zero}$ | ... | $\texttt{Suc}^{i-1}\ \texttt{Zero}$

- Models may be spurious . . . unless IDTs only occur *positively*

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\texttt{nat}^i = \texttt{Zero}$ | $\texttt{Suc Zero}$ | ... | $\texttt{Suc}^{i-1}\ \texttt{Zero}$

- Models may be spurious . . . unless IDTs only occur *positively*
- How to *interpret IDT constructors* of type $\tau \Rightarrow \tau$?

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\mathtt{nat}^i = \mathtt{Zero}$ | $\mathtt{Suc\ Zero}$ | ... | $\mathtt{Suc}^{i-1}\ \mathtt{Zero}$

- Models may be spurious ... unless IDTs only occur *positively*
- How to interpret IDT constructors of type $\tau \Rightarrow \tau$?
  - As partial functions $\tau^i \stackrel{p}{\Longrightarrow} \tau^i$?

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\mathtt{nat}^i = \mathtt{Zero}$ | $\mathtt{Suc\ Zero}$ | ... | $\mathtt{Suc}^{i-1}\ \mathtt{Zero}$

- Models may be spurious . . . unless IDTs only occur *positively*
- How to interpret IDT constructors of type $\tau \Rightarrow \tau$?
  - As partial functions $\tau^i \stackrel{p}{\Longrightarrow} \tau^i$?
  - Or as (total) functions $\tau^{i-1} \Rightarrow \tau^i$?

Inductive Datatypes

Problem: IDTs may require an infinite model

Idea: restrict the constructor depth to obtain *finite approximations* of an IDT

E.g. $\texttt{nat}^i = \texttt{Zero}$ | $\texttt{Suc Zero}$ | ... | $\texttt{Suc}^{i-1}\ \texttt{Zero}$

- Models may be spurious ... unless IDTs only occur *positively*
- How to interpret IDT constructors of type $\tau \Rightarrow \tau$?
  - As partial functions $\tau^i \stackrel{p}{\Longrightarrow} \tau^i$?
  - Or as (total) functions $\tau^{i-1} \Rightarrow \tau^i$?

Non-recursive DTs (e.g. α $\texttt{option}$) could be treated separately.

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)
- `maxvars`: max. number of boolean variables ($0 \cong \infty$)

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)
- `maxvars`: max. number of boolean variables ($0 \cong \infty$)
- `satfile`: name of the SAT solver's input file
- `satformat`: "sat", "cnf", or "defcnf"

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)
- `maxvars`: max. number of boolean variables ($0 \cong \infty$)
- `satfile`: name of the SAT solver's input file
- `satformat`: "sat", "cnf", or "defcnf"
- `resultfile`: name of the SAT solver's output file
- `success`: success message returned by the SAT solver

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)
- `maxvars`: max. number of boolean variables ($0 \cong \infty$)
- `satfile`: name of the SAT solver's input file
- `satformat`: "sat", "cnf", or "defcnf"
- `resultfile`: name of the SAT solver's output file
- `success`: success message returned by the SAT solver
- `command`: system command to execute the SAT solver

refute

Parameters:

- `minsize`: minimal size of the model
- `maxsize`: maximal size of the model ($0 \cong \infty$)
- `maxvars`: max. number of boolean variables ($0 \cong \infty$)
- `satfile`: name of the SAT solver's input file
- `satformat`: "sat", "cnf", or "defcnf"
- `resultfile`: name of the SAT solver's output file
- `success`: success message returned by the SAT solver
- `command`: system command to execute the SAT solver

All parameters can be set globally with `refute_params`.

Future Work

- Theory signatures – to allow user-defined constants and types
- Lazy data structures – to reduce memory requirements
- Types as terms – to have the SAT solver partition the universe
- Binary arithmetic – to reduce the number of boolean variables
- A (simple) built-in SAT solver – to simplify installation
- An external model generator – better performance?