Logical reasoning and programming

First-Order Logic—Equality and model finding

Karel Chvalovský

CIIRC CTU

FOL with equality

We have intentionally ignored the equality predicate. It is the most common predicate used in FOL and hence it deserves a special treatment. There are two main approaches how to deal with it, equality is either

- a binary predicate and its meaning is given by axioms added to our problem, or
- a logical symbol interpreted by the identity relation on the domain.

Note that we have already discussed equality in SMT.

Naïve handling of equality

We can handle equality as a new binary predicate symbol $\sim$ defined by the following axioms:

- $\forall X(X \sim X)$,
- $\forall X \forall Y(X \sim Y \rightarrow Y \sim X)$,
- $\forall X \forall Y \forall Z(X \sim Y \land Y \sim Z \rightarrow X \sim Z)$,
- $\forall X_1 \dots \forall X_n \forall Y_1 \dots \forall Y_n(X_1 \sim Y_1 \land \cdots \land X_n \sim Y_n \rightarrow f(X_1, \dots, X_n) \sim f(Y_1, \dots, Y_n))$,
- $\forall X_1 \dots \forall X_m \forall Y_1 \dots \forall Y_m(X_1 \sim Y_1 \land \cdots \land X_m \sim Y_m \rightarrow (p(X_1, \dots, X_m) \rightarrow p(Y_m, \dots, Y_m)))$.

for every n-ary function symbol f and m-ary predicate symbol p.

However, this is rarely a feasible approach, mainly thanks to the congruence axioms (for function and predicate symbols) and transitivity.

Axiomatic vs. direct approach

We say that a model is normal if the equality predicate is interpreted as the identity relation on its domain.

Example

We can define $X \sim Y$ over $\mathbb{Z}$ by $X \equiv Y \pmod n$ and it clearly satisfies the previous axioms, but the models are non-normal.

In fact, it is impossible to force that all models are normal just by using axioms. For any $\mathcal{M} = (D, i)$, we may add a fresh constant c that will be interpreted exactly as a constant $d \in D$ and we get a new non-normal model.

Theorem

Any set of formulae Γ has a normal model iff Γ has a model satisfying the previous equality axioms.

Proof.

We get a normal model by partitioning the domain into equivalence classes, $[a] = \{\, b \mid b \sim a \,\}$, and use them as the new domain. $\square$

Paramodulation

We say $s \doteqdot t$ if $s = t$ or $t = s$, because order will be important for us later on.

Let $l, l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}$ be literals and s, s' and t be terms. Moreover, $l[s']$ means that the literal l contains s'

$$\frac{\{l_1, \ldots, l_m, s \doteqdot t\} \qquad \{l[s'], l_{m+1}, \ldots, l_{m+n}\}}{\{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}, l[t]\}\sigma}$$

where $\sigma = mgu(s, s')$ and $l[t]$ is the result of replacing an occurrence of s' in $l[s']$ by t. We assume that the input clauses do not share variables (renaming away).

The clause $\{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}, l[t]\}\sigma$ produced by the paramodulation rule is sometimes called the *paramodulant*.

Example

From $\{f(X) = g(X, e)\}$ and $\{p(h(f(0), g(Y, e)), 2)\}$ we can derive $\{p(h(g(0, e), g(Y, e)), 2)\}$ and $\{p(h(f(0), f(Y)), 2)\}$ as paramodulants.

Reflexivity resolution

Clearly, we cannot refute

$$\{\neg X = X\}$$

by the paramodulation rule. Hence we have to add also a rule for such cases. Let $l_1, \ldots, l_m$ be literals and s and t be terms, then the reflexivity resolution rule is

$$\frac{\{l_1, \ldots, l_m, \neg s = t\}}{\{l_1, \ldots, l_m\}\sigma}$$

where $\sigma = mgu(s, t)$.

Completeness of paramodulation

We define $\Gamma \vdash_= \varphi$ similarly to $\vdash$, but, moreover, we can use the paramodulation rule and the reflexivity resolution rule.

Theorem

Let Γ be a set of clauses in the FOL language with equality. Γ is unsatisfiable iff $\Gamma \vdash_= \Box$.

It is possible to produce various restrictions, for example, it is *never necessary to replace a variable by a more complex term* using paramodulation. Loosely speaking, the aim of paramodulation is to make things unifiable, by changing a variable into a complex term we do not improve on this.

However, the most important modification is if we impose orderings on equalities.

Example on equalities (word problem for group theory)

Assume we have axioms

$$\begin{aligned} 1 \cdot X &= X \\ X^{-1} \cdot X &= 1 \\ (X \cdot Y) \cdot Z &= X \cdot (Y \cdot Z) \end{aligned}$$

and we want to know whether $X \cdot Y = Y \cdot X^{-1}$ follows from them?

Example on equalities (word problem for group theory)

Assume we have axioms

$$\begin{aligned} 1 \cdot X &= X \\ X^{-1} \cdot X &= 1 \\ (X \cdot Y) \cdot Z &= X \cdot (Y \cdot Z) \end{aligned}$$

and we want to know whether $X \cdot Y = Y \cdot X^{-1}$ follows from them?

It would be nice to direct axioms, say left to right (or from bigger to smaller), and use them only in this one direction, where two terms are equal if they reduce to the same term (irreducible word).

Example on equalities (word problem for group theory)

Assume we have axioms

$$1 \cdot X = X$$
$$X^{-1} \cdot X = 1$$
$$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$$

and we want to know whether $X \cdot Y = Y \cdot X^{-1}$ follows from them?

It would be nice to direct axioms, say left to right (or from bigger to smaller), and use them only in this one direction, where two terms are equal if they reduce to the same term (irreducible word).

However, this is not sufficient, because we know that, e.g., $Y \cdot (X \cdot X^{-1}) = Y$ holds. Hence we have to add more rules!

Solution

Using the Knuth–Bendix completion we produce the following set of directed rewriting rules (also called reduction rules):

$$\begin{aligned}
1 \cdot X &\succ X \\
X^{-1} \cdot X &\succ 1 \\
(X \cdot Y) \cdot Z &\succ X \cdot (Y \cdot Z) \\
X^{-1} \cdot (X \cdot Y) &\succ Y \\
X \cdot 1 &\succ X \\
1^{-1} &\succ 1 \\
X^{-1-1} &\succ X \\
X \cdot X^{-1} &\succ 1 \\
X \cdot (X^{-1} \cdot Y) &\succ Y \\
(X \cdot Y)^{-1} &\succ Y^{-1} \cdot X^{-1}
\end{aligned}$$

Note that the Knuth–Bendix completion may fail.

Equalities are general

It is possible to express every FOL problem as a problem using only equalities by the following transformation:

$$p(t_1,\ldots,t_n) \text{ becomes } f_p(t_1,\ldots,t_n)=\top,$$
$$\neg p(t_1,\ldots,t_n) \text{ becomes } \neg f_p(t_1,\ldots,t_n)=\top,$$

where $\top$ is a new constant and f_p is a new functional symbol for every predicate p in our original language. Note that f_p and $\top$ are not valid arguments of other terms.

Example

$\{p(X),\neg q(X,g(X,Y))\}$ becomes
$\{f_p(X)=\top,\neg f_q(X,g(X,Y))=\top\}$.

Term orderings

We say that a binary relation $\rightsquigarrow$ is a *rewrite relation*, if it is

- stable under contexts: $s \rightsquigarrow t$ implies $u[s] \rightsquigarrow u[t]$, and
- stable under substitutions: $s \rightsquigarrow t$ implies $s\sigma \rightsquigarrow t\sigma$

where s, t, and u are terms and σ is a substitution.

A *reduction ordering*, denoted $\sqsupset$, is a rewrite relation where $\rightsquigarrow$ is irreflexive, transitive, and well-founded, i.e., there is no infinite strictly-descending sequence $s_1 \sqsupset s_2 \sqsupset \dots$.

A *simplification ordering*, denoted $\succ$, is a reduction ordering where a term is larger than all its proper subterms, i.e., $u[s] \succ s$ if $u[s] \neq s$.

Example

$f(X)$ and $g(Y)$ are $\sqsupset$-incomparable. Let $f(X) \sqsupset g(Y)$ and $\sigma = \{Y \mapsto f(X)\}$, then $f(X) \sqsupset g(f(X))$, and hence $g(f(X)) \sqsupset g(g(f(X)))$, ... Therefore, it is possible that all (non-ground) terms and literals are maximal under a $\sqsupset$.

Useful orderings

Very popular simplification orderings are:

KBO (Knuth–Bendix Ordering)

- uses function symbols weights and precedence to break ties
- produces syntactically smaller terms and is more efficient

LPO (Lexicographic Path Ordering)

- uses function symbols precedence and lexicographic decompositions to break ties
- produces better directions in many cases, e.g., for distributivity

For further details see for example Baader and Nipkow 1998.

Note that the EQP prover used orderings to prove the Robbins conjecture—are the algebras satisfying a given set of axioms exactly Boolean algebras.

Superposition

Although paramodulation rule

$$\frac{\{l_1, \ldots, l_m, s \doteq t\} \qquad \{l[s'], l_{m+1}, \ldots, l_{m+n}\}}{\{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}, l[t]\}\sigma}$$

where $\sigma = mgu(s, s')$, is more efficient than the naïve approach, it has to be further improved to be practical.

We want

- to use only maximal literals (ordered resolution),
- to use only maximal sides of literals (completion),
- s' not to be a variable

and the superposition calculus satisfies all this.

It is used in almost all state-of-the-art automated theorem provers.

How do we show that a formula is not provable?

We have seen several methods that can be used to prove a formula φ from a set of formulae Γ and hence $\Gamma \models \varphi$. However, can we show that

$$\Gamma \not\models \varphi$$

by these methods? Sometimes we can, but it is quite rare, e.g., if we obtain a saturated set.

Note that $\Gamma \not\models \varphi$ is not equivalent to $\Gamma \models \neg\varphi$! For example, $\not\models p(a)$ and $\not\models \neg p(a)$.

A general method is to provide a counterexample. A model of Γ where φ is false, for simplicity assume that φ is a closed formula.

How do we find a counterexample?

We have to check all possible models.

Finite models

For a finite language and a given size of domain, it is possible to check all possible models exhaustively (up to trivial isomorphisms).

Infinite models

Clearly, there are many sets of formulae with only infinite models, for example,

$$\begin{aligned}
&\forall X \neg (X < X),\\
&\forall X \forall Y \forall Z (X < Y \land Y < Z \to X < Z),\\
&\forall X \exists Y (X < Y).
\end{aligned}$$

However, the problem how to generate useful infinite models is widely open. Moreover, the necessity to consider infinite models makes first-order logic algorithmically undecidable (precisely semi-decidable). Fortunately, for many problems finite counterexamples are sufficient.

MACE-style approach

We attempt to generate a finite counterexample iteratively. We try to produce a model of size 1, 2, 3, . . .

The main idea is to produce a grounding of the problem assuming a given cardinality of our model and encode such a grounding as a SAT problem. Using a clever encoding we can significantly reduce the search space; no need to go through all possible models of the given size.

We present some basic techniques used in a model finder called Paradox, see Claessen and Sörensson 2003.

Our example

There is a counterexample for the problem that from

$$\forall X(e \cdot X = X),$$
$$\forall X(X^{-1} \cdot X = e),$$
$$\forall X \forall Y \forall Z(X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z)$$

follows

$$\forall X(X \cdot (X \cdot X) = X).$$

Or equivalently.

Our example

There is a model for

$$\forall X(e \cdot X = X),$$
$$\forall X(X^{-1} \cdot X = e),$$
$$\forall X \forall Y \forall Z(X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z),$$
$$\neg(a \cdot (a \cdot a) = a).$$

We have $\mathcal{M} = (D, i)$, where $D = \{1, 2, 3\}$, $i(e) = 1$, $i(a) = 2$, and

$i(^{-1})$	
1	1
2	3
3	2

$i(\cdot)$	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

Propositional encoding

We are looking for a model $\mathcal{M} = (D, i)$ of a given cardinality, say n, that satisfies a set of clauses Γ. Assume without loss of generality that $D = \{1, \ldots, n\}$. Hence it only remains to generate a function i.

We want to describe i using propositional variables (atoms). For every k-ary

- predicate symbol p in Γ, there is a prop. variable for every
$$p(d_1, \ldots, d_k)$$
where $d_1, \ldots, d_k \in D$.
- function symbol f in Γ, there is a prop. variable for every
$$f(d_1, \ldots, d_k) = d$$
where $d_1, \ldots, d_k, d \in D$.

This is all we need to describe a model.

Our example

Note that our example is a bit confusing—we have only one predicate symbol ($=$), which is very special, because it has the fixed meaning. Still we have atoms for all

$$1 = 1, \quad 1 = 2, \quad 1 = 3, \quad 2 = 1, \quad \ldots, \quad 3 = 2, \quad 3 = 3$$

We also have propositional variables for all

$$\begin{gathered} e = 1, \quad e = 2, \quad e = 3 \\ a = 1, \quad a = 2, \quad a = 3 \\ 1^{-1} = 1, \quad 1^{-1} = 2, \quad 1^{-1} = 3, \quad 2^{-1} = 1, \quad \ldots, \quad 3^{-1} = 3 \\ 1 \cdot 1 = 1, \quad 1 \cdot 1 = 2, \quad 1 \cdot 1 = 3, \quad 1 \cdot 2 = 1, \quad \ldots, \quad 3 \cdot 3 = 3 \end{gathered}$$

Flattening

However, it is impossible to express complex terms like $X \cdot (Y \cdot Z)$ directly in our language. We can only express so called shallow literals:

- $p(X_1, \ldots, X_k)$, or $\neg p(X_1, \ldots, X_k)$,
- $f(X_1, \ldots, X_l) = Y$, or $f(X_1, \ldots, X_l) \neq Y$,
- $X = Y$.

Note that $s \neq t$ is a shortcut for $\neg s = t$.

We do not want $X \neq Y$, because we can transform a clause

$$\{X \neq Y, \varphi(X, Y)\}$$

into

$$\{\varphi(X, X)\}.$$

Note that a clause $\{X \neq Y\}$ is unsatisfiable.

Flattening complex terms

If we have a clause

$$\varphi(t),$$

it is equivalent to

$$\forall X(X = t \to \varphi(X)),$$

which is

$$X \neq t \vee \varphi(X)$$

where X is fresh in $\varphi(t)$. $\varphi(X)$ is produced from $\varphi(t)$ by replacing all (free) occurrences of t by X.

We can repeat this process as long as necessary.

Example

$$\begin{aligned}
&X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \rightsquigarrow \\
&\quad (X \cdot Y) \cdot Z \neq W \vee X \cdot (Y \cdot Z) = W \rightsquigarrow \\
&\quad X \cdot Y \neq V \vee V \cdot Z \neq W \vee X \cdot (Y \cdot Z) = W \rightsquigarrow \\
&\quad X \cdot Y \neq V \vee V \cdot Z \neq W \vee Y \cdot Z \neq U \vee X \cdot U = W.
\end{aligned}$$

Instantiating

For every flattened clause we create three sets of propositional clauses

1. instances — we generate all possible groundings, where we can immediately simplify all groundings containing $d_1 = d_2$ or $d_1 \neq d_2$, for $d_1, d_2 \in D$, based on whether it is true (discard the clause), or not (discard the literal)
2. function definitions — for each k-ary function f and $d, d' \in D$ such that $d \neq d'$, we add
$$\{f(d_1, \ldots, d_k) \neq d, f(d_1, \ldots, d_k) \neq d'\}$$
for every $d_1, \ldots, d_k \in D$.
3. totality definitions — for each k-ary function f, we add
$$\{f(d_1, \ldots, d_k) = 1, f(d_1, \ldots, d_k) = 2, \ldots, f(d_1, \ldots, d_k) = n\}$$
for every $d_1, \ldots, d_k \in D$.

Reducing the number of distinct variables

The number of instances is exponential in the number of distinct variables in a flattened clause.

Term definitions

It is possible to decrease the number of newly introduced variables during flattening for deep terms by using definitions based on constants. From $a \cdot (a \cdot a)$ we can obtain $a \cdot b$, where b is a fresh constant, and define $b = a \cdot a$. It is also possible to introduce definitions for non-ground terms and use definitions across clauses.

Clause splitting

If a clause can be split into parts, where each part contains less distinct variables than the whole clause, then we can decrease the number of distinct variables by introducing a new predicate.

Example

From $\{p(X, Y), q(Y, Z)\}$, we can produce $\{p(X, Y), r(Y)\}, \{\neg r(Y), q(Y, Z)\}$, where r is a fresh predicate.

Isomorphic models

We have $\mathcal{M} = (D, i)$, where $D = \{1, 2, 3\}$, $i(e) = 1$, $i(a) = 2$, and

$i(^{-1})$	
1	1
2	3
3	2

$i(\cdot)$	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

Note that any permutation on elements of D produces an isomorphic model. It makes no sense to look for all of them.

Static symmetry reduction

It is possible to avoid many isomorphic models using the following symmetry reduction technique. If we start to build a model by interpreting a constant c_1, then we can safely assign $i(c_1) = 1$, because no element of D has an assigned meaning. Hence we have

$$\{c_1 = 1\} \text{ instead of } \{c_1 = 1, c_1 = 2, \ldots, c_1 = n\}.$$

Then we can assume that $i(c_2) \in \{1, 2\}$ and $i(c_3) \in \{1, 2, 3\}$, because it has to be interpreted by an element with a meaning, or the first fresh element (if available). However, if $i(c_2) = 1$, then $i(c_3) \in \{1, 2\}$. Or more generally

$$\{c_i \neq k, c_1 = k-1, c_2 = k-1, \ldots c_{i-1} = k-1\}.$$

This can be used also for functions, however, we have to take into account the meaning assigned to elements of D by constants.

Example

Hence $i(e) = 1$ and $i(a) = 2$ in our example, although $i(a) = 3$ works as well.

Other techniques

It is possible to use other techniques like

- pre-processing in SAT—variable and clause elimination, which is incompatible with an incremental search
- finding bounds for $|D|$
  - look for cardinality axioms
  - EPR (effectively proposional), also called Bernays–Schönfinkel–Ramsey class—no function symbols and a quantifier prefix $\exists^*\forall^*$ hence $|D|$ is bounded by the number of constants occurring in the problem; decidable ($\mathrm{NEXPTIME}$-complete)
- use sorts
  - some problems are expressed in a many-sorted language,
  - other problems can be reformulated in a many-sorted language, if we have parts that can be defined independently

Proving in first-order logic (quick summary)

We are interested in the problem $\Gamma \models \varphi$ in FOL, which is an algorithmically undecidable (precisely semi-decidable) problem. Still we can solve many instances by using the following methods:

- we clausify formulae (skolemization,...),
- we extend the resolution calculus to FOL using unification,
- we add direct equality handling,
- we use term orderings.

The current most powerful solvers are based on superposition calculus and we also showed how to use SAT to find small counter-examples.

Note that all these methods are machine-oriented and not particularly suitable for humans (tableaux systems are better). We will use them in the next lecture on proof assistants.

Bibliography I

Baader, Franz and Tobias Nipkow (1998). *Term Rewriting and All That*. Cambridge University Press.

Claessen, Koen and Niklas Sörensson (2003). “New Techniques that Improve MACE-style Finite Model Finding”. In: *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*. Ed. by Peter Baumgartner and Chris Fermüller.

D’Agostino, Marcello et al., eds. (1999). *Handbook of Tableau Methods*. Springer Netherlands. DOI: 10.1007/978-94-017-1754-0.

Harrison, John (Mar. 2009). *Handbook of Practical Logic and Automated Reasoning*. New York: Cambridge University Press, p. 702. URL: http://www.cambridge.org/9780521899574.

Robinson, John Alan and Andrei Voronkov, eds. (2001). *Handbook of Automated Reasoning*. Vol. 1. Elsevier Science.

Tableaux systems

There are many other approaches used to prove formulae in FOL. For example, we have (semantic) tableaux. There are many simple implementations of tableaux in Prolog, e.g., leanT^{A}P or leanCoP, available.

Tableaux systems are also popular in non-classical logics, because

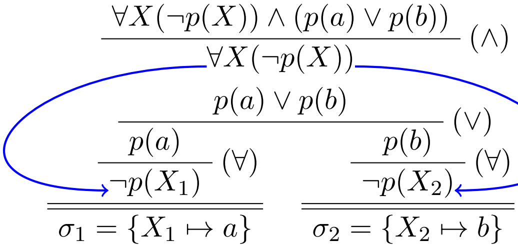
- there is no need for special normal forms like CNF,
    - can be complicated, or
    - even impossible to obtain
- given a semantic meaning of a connective we can usually produce a rule (or rules) in a straightforward way.

Generally, they are relatively easy to produce, in most cases, and still suitable for automated theorem proving. However, the handling of equality is as tricky as in resolution (superposition).

Moreover, they are similar to other proof systems like natural deduction and sequent calculi.

Example

We want to prove $\forall X(\neg p(X)) \rightarrow (\neg p(a) \land \neg p(b))$. We can prove it by showing that $\forall X(\neg p(X))$ and $p(a) \lor p(b)$ are together unsatisfiable.



$$\frac{\forall X(\neg p(X)) \land (p(a) \lor p(b))}{\forall X(\neg p(X))}\ (\land)$$

$$\frac{p(a) \lor p(b)}{\dfrac{\dfrac{p(a)}{\neg p(X_1)}\ (\forall)}{\sigma_1 = \{X_1 \mapsto a\}} \qquad \dfrac{\dfrac{p(b)}{\neg p(X_2)}\ (\forall)}{\sigma_2 = \{X_2 \mapsto b\}}}\ (\lor)$$

Note that we have to use $\forall X(\neg p(X))$ twice.

leanT^{A}P

```
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
   prove(A,[B|UnExp],Lits,FreeV,VarLim).
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
   prove(A,UnExp,Lits,FreeV,VarLim),
   prove(B,UnExp,Lits,FreeV,VarLim).
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
   \+ length(FreeV,VarLim),
   copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
   append(UnExp,[all(X,Fml)],UnExp1),
   prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
prove(Lit,_,[L|Lits],_,_) :-
   (Lit = -Neg; -Lit = Neg) ->
   (unify(Neg,L); prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
   prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```