CS655 Lecture 19 Automated Theorem Proving and Proof Checking 4 April, 2006
Scribe: Nora Sovarel Lecturer: Wes Weimer

Today's lecture is a condensed presentation of automated theorem proving and proof checking. It included logic syntax, theories, satisfiability procedures, mixed theories, theorem proving, proof checking, and SAT-based theorem provers. Automated theorem provers and proof checking are used in practice to rule out infeasible paths, reason about the heap, automatically synthesize programs from specifications, and discover proofs of conjectures.

Automated deduction is defined as logical deduction performed by a machine. It is one of the oldest fields in computer science with results as old as 75 years. Automation efforts span an interval of 40 years, but results are still experimental. Many popular theorem provers and proof assistants are not fully automatic; for exmaple, PVS still needs guidance from a human, but most of the work is automated.

We start with a program and a specification (e.g., "do not derefence null pointers"). The goal is either to prove that the program meets the specification or to find a bug. Proving that the program meets the specification using the language semantics alone is very difficult. Instead, we produce a theorem in a logic (e.g., via verification condition generation), and, in a second step, we use automated theorem proving to reach to the desired goal. Ideally, we want to prove all true things and prove only true things. In theory we cannot have both of these goals and in practice we often obtain neither of them.

1 Logic

The logic we will use is a subset of first-order logic.

Goals (what we want to prove): $G ::= L \mid true \mid G_1 \wedge G_2 \mid H \Rightarrow G \mid \forall x.G$.

Hypotheses: $H ::= L \mid true \mid H_1 \wedge H_2$.

Literals (facts): $L ::= p(E_1, ..., E_k)$. (p is a predicate over expressions like $<$ or $=$)

Expressions: $E ::= n \mid f(E_1, ..., E_m)$. ($f$ is a function like $+$, *, sel, or upd).

2 Theorem Proving

The theorem proving problem is to write an algorithm prove with the following properties:

- soundness: If $prove(G) = true$ then $\models G$.
- completeness: If $\models G$ then $prove(G) = true$.

Since we can not have both, soundness is the one we must have. We want to prove true things and we want to avoid proving false things.

Basic symbolic theorem prover:

$prove(H, true) = true$

$prove(H, G_1 \wedge G_2) = prove(H_1, G) \&\& prove(H_2, G2)$ (The goals can be proved independently.)

$prove(H_1, H_2 \Rightarrow G) = prove(H_1 \wedge H_1, G)$

$prove(H, \forall x.G) = prove(H, G[a/x])$, ($a$ is a *fresh* new variable)

$prove(H, L) = Unsat(H \wedge \neg L)$

We start with prove(H,L), which is equivalent with $H \Rightarrow L$. But, $H = L_1 \wedge ... \wedge L_k$, so prove(H,L) is equivalent with $L_1 \wedge ... \wedge L_k \Rightarrow L$, which is equivalent with $L_1 \wedge ... \wedge L_k \wedge \neg L$.

3 Theory

A theory consists of a set of functions (see expression definition) and predicate simbols (see literals definition) and definitions for the meaning of those symbols.

We define different theories for theory of integers with arithmetic, theory of total orders, theory of lists. For example, addition is not defined in the theory of lists, but we define sel and upd.

A theory is decidable when there is an algorithm deciding whether a formula in a theory with first-order logic is true. The algorithm is called the decision procedure for that theory.

To solve the decidability of a theory problem we will reduce it to the satisfiability problem for that theory. We have to transform the decision formula to a conjunction of literals. A decision procedure for SAT is called a "SAT solver".

3.1 Theory of equality with uninterpreted functions

The symbols used are: $=$, $\neq$, f, g. We don't know what the function symbols (e.g. f, g) mean, but we know if we apply them to the same inputs we will obtain the same outputs.

Reflexivity:

$$\frac{}{A = A}$$

Symmetry:

$$\frac{B = A}{A = B}$$

Transitivity:

$$\frac{A = B \quad B = C}{A = C}$$

Uninterpreted function equality:

$$\frac{A = B}{f(A) = f(B)}$$

For other examples of theories, please see slide 16.

3.2 Mixed theories

In real life we have to deal with symbols from multiple theories. We have a separate SAT procedure for each theory, but building a SAT solver for a combination of theories is much harder. Separating out the terms and separately verifying they are satisfiable it is unsound.

One of the ideas proposed is cooperating SAT procedures. Each SAT procedure works on the literals it understands and it announces all equalities between variables it discovers. Unfortunatelly, there is no obvious way of ordering the SAT procedures application.

A major component of the Nelson-Oppen solution is the Equivalance DAG (E-DAG). We introduce a node for each variable and for each symbol. Nodes cannot be duplicated. Once we dicover an equality we merge the nodes. In summary, the solution involves repeatedly running each procedure, merging the nodes corresponding to equalities announced by the procedure, until there are no more equalities or a contradiction is found. If a contradiction is found then the formula is unsatisfiable. If there are no more equalities then the formula is satisfiable. The theorem prover is relatively complete (with respect to the incomplete SAT procedures).

3.3 Proofs

Theorem provers tell you whether the formula is satisfiable or not. It is useful to extend them to generate a proof. If we can verify the proof then we do not need to trust the theorem prover. This extention allows us to find bugs in the theorem prover and we can use it for proof-carrying code, to extract invariants, or to extract models.

A proof is represented as a tree with leaves as hypotheses or axioms and internal nodes as inference rules. We use a type system to represent the proofs and a type checker to verify them. A simple type system will not work, the proof must be valid and it must prove what we want. A proof for 2+2=4 is valid, but useless for proving that any other program is safe. The solution is to use a dependant type system. Dependant type systems are usually indecidable, but dependant type systems for proof checking are sufficiently restricted that they are decidable.

3.4 SAT-based theorem provers

The idea is to reduce the proof obligation into propositional logic and feed the result to a SAT solver. If it finds a satisfying assignment then the original to-prove may be false: the satisfiability procedures are queried to see if that combination of literals can actually be satisfied. If it can, an extra contrainst is added to prevent the SAt solver from finding that solution again it and is re-run on the augmented problem. If the SAT solver returns no satisfying assignment then the original to-prove goal is true.