Advanced C Programming

Sebastian Hack

hack@cs.uni-sb.de

Christoph Weidenbach

weidenbach@mpi-inf.mpg.de

Winter Term 2008/09

SAARLAND UNIVERSITY

COMPUTER SCIENCE

Why Advanced C?

"Our"

- we need experienced C programmers

"Religious"

- portability
- efficiency
- powerful and flexible

"Real"

- unix
- network software
- embedded systems
- research: graphics, vision, formal methods
- entertainment: games, films

Content I

Content II

Propositional logic

- logic of truth values
- decidable (but NP-complete)
- can be used to describe functions over a finite domain
- important for hardware applications (e.g., model checking)

Syntax

- propositional variables: $P, Q, R \in \Pi$
- logical symbols: $\wedge$ and, $\vee$ or, $\neg$ not, $\top$ true, $\bot$ false
- literals are propositional variables or their negation: P, $\neg P$
- clauses are (posssibly empty) disjunctions of literals: $P \vee \neg Q \vee R$
- clause sets are sets of clauses interpreted as the conjunction of all clauses
- literals, clauses and clause sets are formulas

Semantics

Classical

In classical logic (dating back to Aristoteles) there are “only” two truth values “true” and “false” which we shall denote, respectively, by 1 and 0.

Valuations

A propositional variable has no intrinsic meaning. The meaning of a propositional variable has to be defined by a valuation.

A Π-valuation is a map

$$\mathcal{A} : \Pi \to \{0, 1\}.$$

where $\{0, 1\}$ is the set of truth values.

Truth Value of a Literal, Clause, Clause Set

Given a Π-valuation $\mathcal{A}$, it can be extended to formulas
$\mathcal{A}$: formulas $\rightarrow \{0, 1\}$ inductively as follows:

$$\begin{aligned}
\mathcal{A}(\bot) &= 0 \\
\mathcal{A}(\top) &= 1 \\
\mathcal{A}(P) &= \mathcal{A}(P) \\
\mathcal{A}(\neg P) &= 1 - \mathcal{A}(P) \\
\mathcal{A}(A \vee B) &= \max(\mathcal{A}(A), \mathcal{A}(B)) \\
\mathcal{A}(C \wedge D) &= \min(\mathcal{A}(C), \mathcal{A}(D))
\end{aligned}$$

Models, Validity, and Satisfiability

Validity

F is valid in $\mathcal{A}$ ($\mathcal{A}$ is a model of F; F holds under $\mathcal{A}$):

$$\mathcal{A} \models F \;:\Leftrightarrow\; \mathcal{A}(F) = 1$$

F is valid (or is a tautology):

$$\models F \;:\Leftrightarrow\; \mathcal{A} \models F \text{ for all } \Pi\text{-valuations } \mathcal{A}$$

(Un)Satisfiability

F is called satisfiable if there exists an $\mathcal{A}$ such that $\mathcal{A} \models F$. Otherwise F is called unsatisfiable (or contradictory).

Hence, F is valid iff $\neg F$ is unsatisfiable.
We say that $N \models F$ iff $N \wedge \neg F$ is unsatisfiable.

Checking Unsatisfiability

Every formula F contains only finitely many propositional variables. Obviously, $\mathcal{A}(F)$ depends only on the values of those finitely many variables in F under $\mathcal{A}$.

If F contains n distinct propositional variables, then it is sufficient to check 2^n valuations to see whether F is satisfiable or not $\Rightarrow$ truth table.

So the satisfiability problem is clearly decidable (but, by Cook's Theorem, NP-complete). Nevertheless, in practice, there are (much) better methods than truth tables to check the satisfiability of a formula.

The DPLL Procedure

Goal

Given a propositional formula in CNF (or alternatively, a finite set N of clauses), check whether it is satisfiable (and optionally: output one solution, if it is satisfiable).

Assumption

Clauses contain neither duplicated literals nor complementary literals.

Notation

$\overline{L}$ is the complementary literal of L, i.e., $\overline{P} = \neg P$ and $\overline{\neg P} = P$.

Partial Valuations

Since we will construct satisfying valuations incrementally, we consider partial valuations (that is, partial mappings $\mathcal{A} : \Pi \rightarrow \{0, 1\}$).

Every partial valuation $\mathcal{A}$ corresponds to a set M of literals that does not contain complementary literals, and vice versa:

- $\mathcal{A}(L)$ is true, if $L \in M$.
- $\mathcal{A}(L)$ is false, if $\overline{L} \in M$.
- $\mathcal{A}(L)$ is undefined, if neither $L \in M$ nor $\overline{L} \in M$.

A clause is true under a partial valuation $\mathcal{A}$ (or under a set M of literals) if one of its literals is true; it is false if all its literals are false; otherwise it is undefined.

Unit Clauses

Observation

Let $\mathcal{A}$ be a partial valuation. If the set N contains a clause C, such that all literals but one in C are false under $\mathcal{A}$, then the following properties are equivalent:

- there is a valuation that is a model of N and extends $\mathcal{A}$.
- there is a valuation that is a model of N and extends $\mathcal{A}$ and makes the remaining literal L of C true.

C is called a unit clause; L is called a unit literal.

The Davis-Putnam-Logemann-Loveland Procedure

```
booleanDPLL(literal set M, clause set N) {
    if (all clauses in N are true under M) return true;
    elsif (some clause in N is false under M) return false;
    elsif (N contains unit clause P) return DPLL(M ∪ {P}, N);
    elsif (N contains unit clause ¬P) return DPLL(M ∪ {¬P}, N);
    else {
        let P be some undefined variable in N;
        if (DPLL(M ∪ {¬P}, N)) return true;
        else return DPLL(M ∪ {P}, N);
    }
}
```

Initially, DPLL is called with an empty literal set and the clause set N.

DPLL Iteratively

In practice, there are several changes to the procedure:

- The branching variable is not chosen randomly.
- The algorithm is implemented iteratively;
  the backtrack stack is managed explicitly
  (it may be possible and useful to backtrack more than one level).
- Information is reused by learning.

Formalizing DPLL with Refinements

The DPLL procedure is modelled by a transition relation $\Rightarrow_{\mathrm{DPLL}}$ on a set of states.

States

- *fail*
- $M \parallel N$,

where M is a list of annotated literals and N is a set of clauses.

Annotated literal

- L: deduced literal, due to unit propagation.
- L^{d}: decision literal (guessed literal).

DPLL Rules

Unit Propagate

$M \parallel N \cup \{C \vee L\} \;\Rightarrow_{\mathrm{DPLL}}\; M\,L \parallel N \cup \{C \vee L\}$
if C is false under M and L is undefined under M.

Decide

$M \parallel N \;\Rightarrow_{\mathrm{DPLL}}\; M\,L^{\mathrm{d}} \parallel N$
if L is undefined under M.

Fail

$M \parallel N \cup \{C\} \;\Rightarrow_{\mathrm{DPLL}}\; \mathit{fail}$
if C is false under M and M contains no decision literals.

DPLL Rules

Backjump

$M' \; L^{\mathrm{d}} \; M'' \parallel N \;\; \Rightarrow_{\mathrm{DPLL}} \;\; M' \; L' \parallel N$

if there is some "backjump clause" $C \vee L'$ such that

$N \models C \vee L'$,

C is false under M', and

L' is undefined under M'.

Backtracking

The Backjump rule is always applicable, if the list of literals M contains at least one decision literal and some clause in N is false under M.

There are many possible backjump clauses. One candidate: $\overline{L_1} \vee \ldots \vee \overline{L_n}$, where the L_i are all the decision literals in $M\ L^{\mathrm{d}}\ M'$. (But usually there are better choices.)

DIMACS SAT File Input Format

Syntax

{c <comment>}*
p cnf <number of variables> <number of clauses>
{<clause> 0}*

A <clause> is a sequence of integers from + <number of variables> to − <number of variables>, except 0, separated by blanks.

Example

The clauses $P \lor \neg Q \lor R$, $Q \lor \neg R$ can be coded by the file

```
c first, simple example
p cnf 3 2
1 -2 3 0
2 -3 0
```