Decision Procedures for Verification

Part 1. Propositional Logic (3)

5.11.2013

Viorica Sofronie-Stokkermans

sofronie@uni-koblenz.de

Organization

At the moment:
Lecture: Tue, 14:00-16:00
Exercise: Thu, 12:30-14:00 (can be changed to 12:00-14:00)

Question 1:
Would it be better to switch lecture/exercises?
Answer: Starting from next week we switch lecture/exercises

Schedule starting from next week:
Exercises: Tue, 14:00 s.t. -16:00
Lecture: Thu, 12:00 s.t.-14:00

Question 2:
Is Thu, 10:00-12:00 a better time than Thu. 12:30-14:00?
Answer: No, inconvenient for some of the participants

Last time

1.1 Syntax

- Language
  - propositional variables
  - logical symbols

    $\Rightarrow$ Boolean combinations
- Propositional Formulae

1.2 Semantics

- Valuations
- Truth value of a formula in a valuation
- Models, Validity, and Satisfiability
- Entailment and Equivalence

Canonical forms

- CNF and DNF
- Computing CNF/DNF by rewriting the formulae
- Structure-Preserving Translation for CNF
- Optimized translation using polarity

Decision Procedures for Satisfiability

- Simple Decision Procedures

  truth table method — Logik f. Informatiker

  Discrete Algebraic Structures

- The Resolution Procedure — last time

- The Davis-Putnam-Logemann-Loveland Algorithm — today

1.7 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).

Satisfiability of Clause Sets

$\mathcal{A} \models N$ if and only if $\mathcal{A} \models C$ for all clauses C in N.

$\mathcal{A} \models C$ if and only if $\mathcal{A} \models L$ for some literal $L \in C$.

Partial Valuations

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

We start with an empty valuation and try to extend it step by step to all variables occurring in N.

If $\mathcal{A}$ is a partial valuation, then literals and clauses can be true, false, or undefined under $\mathcal{A}$.

A clause is true under $\mathcal{A}$ if one of its literals is true; it is false (or "conflicting") if all its literals are false; otherwise it is undefined (or "unresolved").

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.

Pure Literals

One more observation:

Let $\mathcal{A}$ be a partial valuation and P a variable that is undefined under $\mathcal{A}$. If P occurs only positively (or only negatively) in the unresolved clauses in N, 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 assigns true (false) to P.

P is called a pure literal.

The Davis-Putnam-Logemann-Loveland Proc.

```
boolean DPLL(clause set N, partial valuation 𝒜) {
    if (all clauses in N are true under 𝒜) return true;
    elsif (some clause in N is false under 𝒜) return false;
    elsif (N contains unit clause P) return DPLL(N, 𝒜 ∪ {P ↦ 1});
    elsif (N contains unit clause ¬P) return DPLL(N, 𝒜 ∪ {P ↦ 0});
    elsif (N contains pure literal P) return DPLL(N, 𝒜 ∪ {P ↦ 1});
    elsif (N contains pure literal ¬P) return DPLL(N, 𝒜 ∪ {P ↦ 0});
    else {
        let P be some undefined variable in N;
        if (DPLL(N, 𝒜 ∪ {P ↦ 0})) return true;
        else return DPLL(N, 𝒜 ∪ {P ↦ 1});
    }
}
```

The Davis-Putnam-Logemann-Loveland Proc.

Initially, DPLL is called with the clause set N and with an empty partial valuation $\mathcal{A}$.

The Davis-Putnam-Logemann-Loveland Proc.

In practice, there are several changes to the procedure:

The pure literal check is often omitted (it is too expensive).

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).

DPLL Iteratively

An iterative (and generalized) version:

```
status = preprocess();
if (status != UNKNOWN) return status;
while(1) {
    decide_next_branch();
    while(1) {
        status = deduce();
        if (status == CONFLICT) {
            blevel = analyze_conflict();
            if (blevel == 0) return UNSATISFIABLE;
            else backtrack(blevel); }
        else if (status == SATISFIABLE) return SATISFIABLE;
        else break;
    }
}
```

DPLL Iteratively

`preprocess()`

preprocess the input (as far as it is possible without branching);
return CONFLICT or SATISFIABLE or UNKNOWN.

`decide_next_branch()`

choose the right undefined variable to branch;
decide whether to set it to 0 or 1;
increase the backtrack level.

DPLL Iteratively

`deduce()`

make further assignments to variables (e.g., using the unit clause rule) until a satisfying assignment is found, or until a conflict is found, or until branching becomes necessary; return CONFLICT or SATISFIABLE or UNKNOWN.

DPLL Iteratively

`analyze_conflict()`

check where to backtrack.

`backtrack(blevel)`

backtrack to `blevel`;

flip the branching variable on that level;

undo the variable assignments in between.

Branching Heuristics

Choosing the right undefined variable to branch is important for efficiency, but the branching heuristics may be expensive itself.

State of the art: use branching heuristics that need not be recomputed too frequently.

In general: choose variables that occur frequently.

The Deduction Algorithm

For applying the unit rule, we need to know the number of literals in a clause that are not false.

Maintaining this number is expensive, however.

The Deduction Algorithm

Better approach: "Two watched literals":

In each clause, select two (currently undefined) "watched" literals.

For each variable P, keep a list of all clauses in which P is watched and a list of all clauses in which $\neg P$ is watched.

If an undefined variable is set to 0 (or to 1), check all clauses in which P (or $\neg P$) is watched and watch another literal (that is true or undefined) in this clause if possible.

Watched literal information need not be restored upon backtracking.

Conflict Analysis and Learning

Goal: Reuse information that is obtained in one branch in further branches.

Method: Learning:

> If a conflicting clause is found, use the resolution rule to derive a new clause and add it to the current set of clauses.

Problem: This may produce a large number of new clauses; therefore it may become necessary to delete some of them afterwards to save space.

Backjumping

Related technique:
non-chronological backtracking ("backjumping"):

If a conflict is independent of some earlier branch, try to skip that over that backtrack level.

Restart

Runtimes of DPLL-style procedures depend extremely on the choice of branching variables.

If no solution is found within a certain time limit, it can be useful to restart from scratch with another choice of branchings (but learned clauses may be kept).

A succinct formulation

State: $M||F$,

where:

- M partial assignment (sequence of literals),

  some literals are annotated (L^d: decision literal)

- F clause set.

A succinct formulation

UnitPropagation

$M||F, C \vee L \Rightarrow M, L||F, C \vee L$ if $M \models \neg C$, and L undef. in M

Decide

$M||F \Rightarrow M, L^d||F$ if L or $\neg L$ occurs in F, L undef. in M

Fail

$M||F, C \Rightarrow \text{Fail}$ if $M \models \neg C$, M contains no decision literals

Backjump

$$M, L^d, N||F \Rightarrow M, L'||F \quad \text{if} \begin{cases} \text{there is some clause } C \vee L' \text{ s.t.:} \\ F \models C \vee L', M \models \neg C, \\ L' \text{ undefined in } M \\ L' \text{ or } \neg L' \text{ occurs in } F. \end{cases}$$

Example

Assignment:	Clause set:	
$\emptyset$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
P_1	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp)
$P_1 P_2$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
$P_1 P_2 P_3$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp)
$P_1 P_2 P_3 P_4$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
$P_1 P_2 P_3 P_4 P_5$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp)
$P_1 P_2 P_3 P_4 P_5 \neg P_6$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Backtrack)
$P_1 P_2 P_3 P_4 \neg P_5$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	...

DPLL with learning

The DPLL system with learning consists of the four transition rules of the Basic DPLL system, plus the following two additional rules:

Learn

$M||F \Rightarrow M||F, C$ if all atoms of C occur in F and $F \models C$

Forget

$M||F, C \Rightarrow M||F$ if $F \models C$

In these two rules, the clause C is said to be learned and forgotten, respectively.

Further Information

The ideas described so far heve been implemented in the SAT checker Chaff.

Further information:
Lintao Zhang and Sharad Malik:
The Quest for Efficient Boolean Satisfiability Solvers,
Proc. CADE-18, LNAI 2392, pp. 295–312, Springer, 2002.

Applications

- A toy example (sudoku)
- Scheduling
- Verification

Sudoku

1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Idea: $p^d_{i,j} =$ true iff the value of square i, j is d

For example: $p^8_{3,5} =$ true

Sudoku

1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Coding SUDOKU by propositional clauses:

- Concrete values result in units: $p^d_{i,j}$.
- For every value, column we generate: $\neg p^d_{i,j} \vee \neg p^d_{i,k}$ (if j

  Accordingly for all rows and 3×3 boxes.
- For every square we generate: $p^1_{i,j} \vee \ldots p^9_{i,j}$.

  For every two different values d, d', and every square we generate: $\neg p^d_{i,j} \vee \neg p^{d'}_{i,j}$.
- For every value d and every column we generate:

  $p^d_{i,1} \vee \ldots p^d_{i,9}$.

  Accordingly for all rows and 3×3 boxes.

Sudoku

1								1	
2	4								
3		2							
4					5		4		7
5			8				3		
6			1		9				
7	3			4			2		
8		5		1					
9				8		6			

Set of clauses satisfiable $\Leftrightarrow$ Sudoku has a solution

Let $\mathcal{A}$ be a satisfying assignment

$\mathcal{A}(p_{i,j}^k) = 1$ iff a k appears in line i, column j.

Scheduling

Example: A simple scheduling problem

In a school there are three teachers with the following specialization combinations:

Müller	Mathematics
Schmidt	German
Körner	Mathematics, German

	Group a	Group b
8:00– 8:50	Mathematics	German
9:00– 9:50	German	German
10:00–10:50	Math	Mathematics

Each teacher must teach at least two classes.

Scheduling

Müller	Mathematics
Schmidt	German
Körner	Mathematics, German

	Group a	Group b
1) 8:00– 8:50	Mathematics	German
2) 9:00– 9:50	German	German
3)10:00–10:50	Math	Mathematics

Modeling:

Propositional variables: $P_{s,k,N,f}$ 'Teacher N teaches subject f in group k in time slot s'

Scheduling

Müller	Mathematics
Schmidt	German
Körner	Mathematics, German

	Group a	Group b
1) 8:00– 8:50	Mathematics	German
2) 9:00– 9:50	German	German
3)10:00–10:50	Math	Mathematics

Modeling:

Propositional variables: $P_{s,k,N,f}$ 'Teacher N teaches subject f in group k in time slot s'

Rules: $(P_{1,a,M,m} \vee P_{1,a,K,m}) \wedge (P_{1,b,S,d} \vee P_{1,b,K,d})$

$$(P_{2,a,S,d} \vee P_{2,a,K,d}) \wedge (P_{2,b,S,d} \vee P_{2,b,K,d})$$

$$(P_{3,a,M,m} \vee P_{3,a,K,m}) \wedge (P_{3,b,S,d} \vee P_{3,a,K,d})$$

$$\neg(P_{1,a,K,m} \wedge P_{1,b,K,d}) \wedge \neg(P_{2,a,K,d} \wedge P_{2,b,K,d}) \wedge \neg(P_{2,a,S,d} \wedge P_{2,b,S,d}) \wedge$$

$$\neg(P_{3,a,K,m} \wedge P_{3,b,K,m}) \wedge (P_{1,a,M,m} \wedge P_{1,b,M,m}) \ldots$$

Program Verification

- Bounded model checking
- Model checking
- Invariant checking/generation
- Abstraction

Finite-state systems

- X finite set of variables, V finite set of possible values for the variables
  p^i_{xv} (in the i-th step x takes value v)
- Other propositional variables q_k, $k \in K$
- Transitions (variables change their name)
  $Tr(i, i+1) := \bigwedge_{j=1}^{n} p^{i+1}_{x_j v i+1_j} \wedge \bigwedge_k q^{i+1}_k$
  (where $vi+1_j$, q^{i+1}_k suitably computed)

$F(p^k_{x_1, v^k_1}, \dots, p^k_{x_n, v^k_n}, ...)$ property of assignments

Bounded model checking:

$$\bigwedge_{j=1}^{n} p^1_{x_j, v_j} \wedge \bigwedge q^1_k \wedge Tr(1,2) \wedge \dots \wedge Tr(k-1, k) \wedge \neg F(p^k_{x_1, v^k_1}, \dots, p^k_{x_n, v^k_n}, ...)$$

Example

Question: Does BubbleSort return a sorted array?

```
int [] BubbleSort(int[] a) {
    int i, j, t;
    for (i := |a| − 1; i > 0; i := i − 1) {
        for (j := 0; j < i; j := j + 1) {
            if (a[j] > a[j + 1]){t := a[j];
                                 a[j] := a[j + 1];
                                 a[j + 1] := t};
}} return a}
```

Example

Question: Does BubbleSort return a sorted array?

```
int [] BubbleSort(int[] a) {
    int i, j, t;
    for (i := |a| − 1; i > 0; i := i − 1) {
        for (j := 0; j < i; j := j + 1) {
            if (a[j] > a[j + 1]){t := a[j];
                                 a[j] := a[j + 1];
                                 a[j + 1] := t};
}} return a}
```

Simpler question:

$|a| = 3$; $a[0]=7$, $a[1]=9$, $a[2]=4$

does BubbleSort applied to this array return a sorted array?

Encoding in propositional logic:

- p_{ij}^k (at step k, $a[i] = k$)

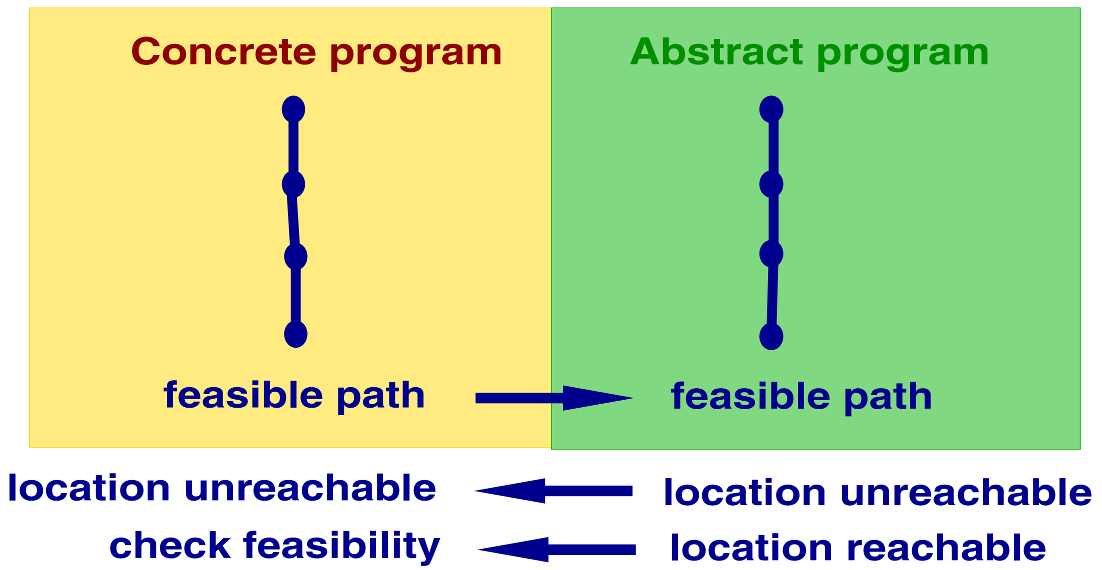
  Examples: $p_{07}^1, p_{19}^1, p_{24}^1$
- gt_{ij}^k (at step k, $a[i] > a[j]$)

  Examples: $gt_{10}^1, \neg gt_{01}^1, gt_{02}^1, \neg gt_{20}^1, \ldots$

Model updates with new propositional variables

(complicated; not very expressive)

Abstraction-Based Verification



$\Downarrow$

conjunction of constraints: $\phi(1) \wedge Tr(1,2) \wedge \cdots \wedge Tr(n-1,n) \wedge \neg \mathsf{safe}(n)$

- satisfiable: feasible path
- unsatisfiable: refine abstract program s.t. the path is not feasible

Tools for SAT checking

http://www.satcompetition.org/

Examples of SAT solvers:

MiniSat: http://minisat.se/

MathSAT: http://mathsat.fbk.eu/publications.html (much more)

zChaff: http://www.princeton.edu/ chaff/zchaff.html

Example of use

Tools for SAT checking

Resolution-based theorem provers:

E: `http://www4.informatik.tu-muenchen.de/ schulz/E/E.html`

SPASS: `http://www.spass-prover.org/`

Vampire: `http://www.vprover.org/`

. . . full power for first-order logic (with equality)

Part 2: First-Order Logic

First-order logic

- formalizes fundamental mathematical concepts
- is expressive (Turing-complete)
- is not too expressive
  (e. g. not axiomatizable: natural numbers, uncountable sets)
- has a rich structure of decidable fragments
- has a rich model and proof theory

First-order logic is also called (first-order) predicate logic.

2.1 Syntax

Syntax:

- non-logical symbols (domain-specific)
  $\Rightarrow$ terms, atomic formulas

- logical symbols (domain-independent)
  $\Rightarrow$ Boolean combinations, quantifiers

Signature

A signature

$$\Sigma = (\Omega, \Pi),$$

fixes an alphabet of non-logical symbols, where

- Ω is a set of function symbols f with arity $n \geq 0$, written f/n,
- Π is a set of predicate symbols p with arity $m \geq 0$, written p/m.

If $n = 0$ then f is also called a constant (symbol).
If $m = 0$ then p is also called a propositional variable.
We use letters P, Q, R, S, to denote propositional variables.

Signature

Refined concept for practical applications:
many-sorted signatures (corresponds to simple type systems in programming languages).

Most results established for one-sorted signatures extend in a natural way to many-sorted signatures.

Many-sorted Signature

A many-sorted signature

$$\Sigma = (S, \Omega, \Pi),$$

fixes an alphabet of non-logical symbols, where

- S is a set of sorts,
- Ω is a set of function symbols f with arity $a(f) = s_1 \ldots s_n \to s$,
- Π is a set of predicate symbols p with arity $a(p) = s_1 \ldots s_m$

where $s_1, \ldots, s_n, s_m, s$ are sorts.

Variables

Predicate logic admits the formulation of abstract, schematic assertions. (Object) variables are the technical tool for schematization.

We assume that

$$X$$

is a given countably infinite set of symbols which we use for (the denotation of) variables.

Variables

Predicate logic admits the formulation of abstract, schematic assertions. (Object) variables are the technical tool for schematization.

We assume that

$$X$$

is a given countably infinite set of symbols which we use for (the denotation of) variables.

Many-sorted case:

We assume that for every sort $s \in S$, X_s is a given countably infinite set of symbols which we use for (the denotation of) variables of sort s.

Terms

Terms over Σ (resp., Σ-terms) are formed according to these syntactic rules:

$$
\begin{array}{lllll}
t, u, v & ::= & x & , x \in X & \text{(variable)} \\
 & \mid & f(s_1, \ldots, s_n) & , f/n \in \Omega & \text{(functional term)}
\end{array}
$$

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X).

A term not containing any variable is called a ground term.

By T_Σ we denote the set of Σ-ground terms.

Terms

Terms over Σ (resp., Σ-terms) are formed according to these syntactic rules:

$$
\begin{array}{rclll}
t, u, v & ::= & x & , x \in X & \text{(variable)} \\
 & \mid & f(t_1, ..., t_n) & , f/n \in \Omega & \text{(functional term)}
\end{array}
$$

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X).
A term not containing any variable is called a ground term.
By T_Σ we denote the set of Σ-ground terms.

Many-sorted case:

a variable $x \in X_s$ is a term of sort s

if $a(f) = s_1 \ldots s_n \to s$, and t_i are terms of sort s_i, $i = 1, \ldots, n$ then $f(t_1, ..., t_n)$ is a term of sort s.

Terms

In other words, terms are formal expressions with well-balanced brackets which we may also view as marked, ordered trees.

The markings are function symbols or variables.

The nodes correspond to the subterms of the term.

A node v that is marked with a function symbol f of arity n has exactly n subtrees representing the n immediate subterms of v.

Atoms

Atoms (also called atomic formulas) over Σ are formed according to this syntax:

$$\begin{array}{lcll} A, B & ::= & p(t_1, ..., t_m) & , \; p/m \in \Pi \\ \Big[& \mid & (t \approx t') & \text{(equation)} \quad \Big] \end{array}$$

Whenever we admit equations as atomic formulas we are in the realm of first-order logic with equality. Admitting equality does not really increase the expressiveness of first-order logic, (cf. exercises). But deductive systems where equality is treated specifically can be much more efficient.

Atoms

Atoms (also called atomic formulas) over Σ are formed according to this syntax:

$$\begin{array}{llll} A, B & ::= & p(t_1, ..., t_m) & , \; p/m \in \Pi \\ \Big[& \mid & (t \approx t') & \text{(equation)} \quad \Big] \end{array}$$

Whenever we admit equations as atomic formulas we are in the realm of first-order logic with equality. Admitting equality does not really increase the expressiveness of first-order logic, (cf. exercises). But deductive systems where equality is treated specifically can be much more efficient.

Many-sorted case:

If $a(p) = s_1 \ldots s_m$, we require that t_i is a term of sort s_i for $i = 1, \ldots, m$.

Literals

$$
\begin{array}{lcll}
L & ::= & A & \text{(positive literal)} \\
 & | & \neg A & \text{(negative literal)}
\end{array}
$$

Clauses

$$
\begin{array}{lcll}
C, D & ::= & \bot & \text{(empty clause)} \\
 & \mid & L_1 \vee \ldots \vee L_k, \;\; k \geq 1 & \text{(non-empty clause)}
\end{array}
$$

General First-Order Formulas

$F_\Sigma(X)$ is the set of first-order formulas over Σ defined as follows:

$$
\begin{array}{llll}
F, G, H & ::= & \bot & \text{(falsum)} \\
 & \mid & \top & \text{(verum)} \\
 & \mid & A & \text{(atomic formula)} \\
 & \mid & \neg F & \text{(negation)} \\
 & \mid & (F \wedge G) & \text{(conjunction)} \\
 & \mid & (F \vee G) & \text{(disjunction)} \\
 & \mid & (F \rightarrow G) & \text{(implication)} \\
 & \mid & (F \leftrightarrow G) & \text{(equivalence)} \\
 & \mid & \forall x F & \text{(universal quantification)} \\
 & \mid & \exists x F & \text{(existential quantification)}
\end{array}
$$

Notational Conventions

We omit brackets according to the following rules:

- $\neg \quad >_p \quad \wedge \quad >_p \quad \vee \quad >_p \quad \rightarrow \quad >_p \quad \leftrightarrow$
  (binding precedences)
- $\vee$ and $\wedge$ are associative and commutative
- $\rightarrow$ is right-associative

$Qx_1, \ldots, x_n\, F$ abbreviates $Qx_1 \ldots Qx_n\, F$.

Notational Conventions

We use infix-, prefix-, postfix-, or mixfix-notation with the usual operator precedences.

Examples:

$s + t * u$	for	$+(s, *(t, u))$
$s * u \leq t + v$	for	$\leq (*(s, u), +(t, v))$
$-s$	for	$-(s)$
0	for	$0()$

Example: Peano Arithmetic

Signature:

$$\Sigma_{PA} = (\Omega_{PA},\ \Pi_{PA})$$
$$\Omega_{PA} = \{0/0,\ +/2,\ */2,\ s/1\}$$
$$\Pi_{PA} = \{\leq/2,\ </2\}$$
$$+,\ *,\ <,\ \leq \text{ infix; } * \ >_p\ +\ >_p\ <\ >_p\ \leq$$

Examples of formulas over this signature are:

$$\forall x, y(x \leq y \leftrightarrow \exists z(x + z \approx y))$$
$$\exists x \forall y(x + y \approx y)$$
$$\forall x, y(x * s(y) \approx x * y + x)$$
$$\forall x, y(s(x) \approx s(y) \rightarrow x \approx y)$$
$$\forall x \exists y(x < y \land \neg\exists z(x < z \land z < y))$$

Remarks About the Example

We observe that the symbols $\leq$, $<$, 0, s are redundant as they can be defined in first-order logic with equality just with the help of $+$. The first formula defines $\leq$, while the second defines zero. The last formula, respectively, defines s.

Eliminating the existential quantifiers by Skolemization (cf. below) reintroduces the "redundant" symbols.

Consequently there is a *trade-off* between the complexity of the quantification structure and the complexity of the signature.

Example: Specifying LISP lists

Signature:

$\Sigma_{\text{Lists}} = (\Omega_{\text{Lists}}, \Pi_{\text{Lists}})$

$\Omega_{\text{Lists}} = \{\text{car}/1, \text{cdr}/1, \text{cons}/2\}$

$\Pi_{\text{Lists}} = \emptyset$

Examples of formulae:

$\forall x, y \;\; \text{car}(\text{cons}(x, y)) \approx x$

$\forall x, y \;\; \text{cdr}(\text{cons}(x, y)) \approx y$

$\forall x \quad \text{cons}(\text{car}(x), \text{cdr}(x)) \approx x$

Many-sorted signatures

Example:

Signature

$S = \{\text{array}, \text{index}, \text{element}\}$ set of sorts

$\Omega = \{\text{read}, \text{write}\}$

$$a(\text{read}) = \text{array} \times \text{index} \to \text{element}$$
$$a(\text{write}) = \text{array} \times \text{index} \times \text{element} \to \text{array}$$

$\Pi = \emptyset$

$X = \{X_s \mid s \in S\}$

Examples of formulae:

$\forall x : \text{array} \ \ \forall i : \text{index} \ \ \forall j : \text{index} \ \ (i \approx j \to \text{write}(x, i, \text{read}(x, j)) \approx x)$

$\forall x : \text{array} \ \forall y : \text{array} \ \ (x \approx y \leftrightarrow \forall i : \text{index} \ \ (\text{read}(x, i) \approx \text{read}(y, i)))$