Predicate Logic

- Invented by Gottlob Frege (1848–1925).
- Predicate Logic is also called “first-order logic”.

“Every good mathematician is at least half a philosopher, and every good philosopher is at least half a mathematician.”
Motivation

There are some kinds of human reasoning that we can’t do in propositional logic.

- For example:
  
  Every person likes ice cream.
  Billy is a person.
  Therefore, Billy likes ice cream.

- In propositional logic, the best we can do is \( A \land B \Rightarrow C \), which isn’t a tautology.
  - We’ve lost the internal structure.

- We need to be able to refer to objects.
- We want to symbolize both a claim and the object about which the claim is made.
- We also need to refer to relations between objects,
  - as in “Waterloo is west of Toronto”.
- If we can refer to objects, we also want to be able to capture the meaning of every and some of.
- The predicates and quantifiers of predicate logic allow us to capture these concepts.
**Apt-pet**

- An apartment pet is a pet that is small
- Dog is a pet
- Cat is a pet
- Elephant is a pet
- Dogs and cats are small.
- Some dogs are cute
- Each dog hates some cat
- Fido is a dog

\[
\forall x \ small(x) \land pet(x) \Rightarrow aptPet(x) \\
\forall x \ dog(x) \Rightarrow pet(x) \\
\forall x \ cat(x) \Rightarrow pet(x) \\
\forall x \ elephant(x) \Rightarrow pet(x) \\
\forall x \ dog(x) \Rightarrow small(x) \\
\forall x \ cat(x) \Rightarrow small(x) \\
\exists x \ dog(x) \land cute(x) \\
\forall x \ dog(x) \Rightarrow \exists y \ cat(y) \land hates(x, y) \\
dog(fido)
\]

---

**Quantifiers**

- Universal quantification (\(\forall\)) corresponds to finite or infinite conjunction of the application of the predicate to all elements of the domain.
- Existential quantification (\(\exists\)) corresponds to finite or infinite disjunction of the application of the predicate to all elements of the domain.
- Relationship between \(\forall\) and \(\exists\):
  - \(\exists x. P(x)\) is the same as \(\neg \forall x. \neg P(x)\)
  - \(\forall x. P(x)\) is the same as \(\neg \exists x. \neg P(x)\)
Functions

- Consider how to formalize:
  - Mary’s father likes music
  - One possible way: $\exists f(x, \text{Mary}) \land \text{Likes}(x, \text{Music})$
  - which means: Mary has at least one father and he likes music.

- We’d like to capture the idea that Mary only has one father.
  - We use functions to capture the single object that can be in relation to another object.
  - Example: $\text{Likes}(\text{father(Mary)}, \text{Music})$

- We can also have $n$-ary functions.

Predicate Logic

- syntax (well-formed formulas)
- semantics
- proof theory
  - axiom systems
  - natural deduction
  - sequent calculus
  - resolution principle
Predicate Logic: Syntax

The syntax of predicate logic consists of:
- constants
- variables \( x, y, \ldots \)
- functions
- predicates
- logical connectives
- quantifiers
- punctuations: , . ( )

Predicate Logic: Syntax

**Definition.** Terms are defined inductively as follows:

- **Base cases**
  - Every constant is a term.
  - Every variable is a term.
- **Inductive cases**
  - If \( t_1, t_2, t_3, \ldots, t_n \) are terms then \( f(t_1, t_2, t_3, \ldots, t_n) \) is a term, where \( f \) is an \( n \)-ary function.
- Nothing else is a term.
Predicate Logic

- syntax

**Definition.** Well-formed formulas (wffs) are defined inductively as follows:

- **Base cases:**
  - \( P(t_1, t_2, t_3, \ldots, t_n) \) is a wff, where \( t_i \) is a term, and \( P \) is an \( n \)-ary predicate. These are called atomic formulas.

- **Inductive cases:**
  - If \( A \) and \( B \) are wffs, then so are \( \neg A, A \land B, A \lor B, A \Rightarrow B, A \Leftrightarrow B \)
  - If \( A \) is a wff, so is \( \exists x. A \)
  - If \( A \) is a wff, so is \( \forall x. A \)

Nothing else is a wff.

We often omit the brackets using the same precedence rules as propositional logic for the logical connectives.

---

Scope and Binding of Variables (I)

- Variables occur both in nodes next to quantifiers and as leaf nodes in the parse tree.
- A variable \( x \) is **bound** if starting at the leaf of \( x \), we walk up the tree and run into a node with a quantifier and \( x \).
- A variable \( x \) is **free** if, starting at the leaf of \( x \), we walk up the tree and don’t run into a node with a quantifier and \( x \).

\[
\forall x. (\forall x. (P(x) \land Q(x))) \Rightarrow (\neg P(x) \lor Q(y))
\]
Scope and Binding of Variables (I)

The **scope** of a variable $x$ is the subtree starting at the node with the variable and its quantifier (where it is bound) minus any subtrees with $\forall x$ or $\exists x$ at their root.

Example:

A wff is **closed** if it contains no free occurrences of any variable.

$$\forall x (\forall x ((P(x) \land Q(x)))) \Rightarrow (\neg P(x) \lor Q(y))$$

Scope and Binding of Variables

$$\forall x ((P(x) \Rightarrow Q(x)) \land S(x,y))$$

Parsing tree:

This is an open formula!

interpreted with

interpreted with

interpreted with

bound variables

free variable

scope of $\forall x$
Scope and Binding of Variables

\[ \forall x((\exists x(P(x) \Rightarrow Q(x))) \land S(x,y)) \]

Parsing tree:

This is an open formula!

Substitution

Variables are place holders.

- Given a variable \( x \), a term \( t \) and a formula \( P \), we define \( P[t/x] \) to be the formula obtained by replacing each free occurrence of variable \( x \) in \( P \) with \( t \).

- We have to watch out for variable captures in substitution.
Substitution

In order not to mess up with the meaning of the original formula, we have the following restrictions on substitution.

- Given a term $t$, a variable $x$ and a formula $P$,
  "$t$ is not free for $x$ in $P$"
  if
  - $x$ in a scope of $\forall y$ or $\exists y$ in $A$; and
  - $t$ contains a free variable $y$.
- Substitution $P[t/x]$ is allows only if $t$ is free for $x$ in $P$.

\[ \forall y (\text{mom}(x) \land \text{dad}(f(y))) \equiv \forall z (\text{mom}(x) \land \text{dad}(f(z))) \]

But

$$(\forall y (\text{mom}(x) \land \text{dad}(y)))[f(y)/x] = \forall y (\text{mom}(f(y)) \land \text{dad}(f(y)))$$

$$(\forall z (\text{mom}(x) \land \text{dad}(z)))[f(y)/x] = \forall z (\text{mom}(f(y)) \land \text{dad}(f(z)))$$

[f(y)/x] not allowed since meaning of formulas messed up.
Predicate Logic: Semantics

- Recall that a semantics is a mapping between two worlds.
- A model for predicate logic consists of:
  - a non-empty domain of objects: $D_i$
  - a mapping, called an interpretation that associates the terms of the syntax with objects in a domain
- It’s important that $D_i$ be non-empty, otherwise some tautologies wouldn’t hold such as $(\forall x.A(x)) \Rightarrow (\exists x.A(x))$

Interpretations (Models)

- a fixed element $c' \in D_i$ to each constant $c$ of the syntax
- an $n$-ary function $f' : D_i^n \rightarrow D_i$ to each $n$-ary function, $f$, of the syntax
- an $n$-ary relation $R' \subseteq D_i^n$ to each $n$-ary predicate, $R$, of the syntax
Example of a Model

- Let’s say our syntax has a constant $c$, a function $f$ (unary), and two predicates $P$, and $Q$ (both binary).

Example: $P(c, f(c))$

In our model, choose the domain to be the natural numbers

- $I(c)$ is 0.
- $I(f)$ is suc, the successor function.
- $I(P)$ is `<`
- $I(Q)$ is `=`

Example of an Model

What’s the meaning of $P(c, f(c))$ in this model?

$I(P(c, f(c))) = \forall \chi. (I(\chi)) < (I(f)\chi)$

- $= 0 < suc(I(\chi))$
- $= 0 < suc(0)$
- $= 0 < 1$

Which is true.
Valuations

Definition.
A valuation \( v \), in an interpretation \( I \), is a function from the terms to the domain \( D_I \) such that:

- \( v(c) = I(c) \)
- \( v(f(t_1, \ldots, t_n)) = f'(v(t_1), \ldots, v(t_n)) \)
- \( v(x) \in D_I \), i.e., each variable is mapped onto some element in \( D_I \).

Example of a Valuation

- \( D_I \) is the set of Natural Numbers
- \( g \) is the function \( + \)
- \( h \) is the function \( suc \)
- \( c \) (constant) is 3
- \( y \) (variable) is 1

\[
v(g(h(c), y)) = v(h(c)) + v(y) \\
= suc(v(c)) + 1 \\
= suc(3) + 1 \\
= 5
\]
Workout

- $D_i$ is the set of Natural Numbers
- $g$ is the function $+$
- $h$ is the function $suc$
- $c$ (constant) is 3
- $y$ (variable) is 1

$v(h(h(g(h(y),g(h(y),h(c)))),))) = ?$
## Workout

Interpret the following formulas with respect to the world (model) in the previous page.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{On}(A, Fl) \Rightarrow \text{Clear}(B)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B) \land \text{Clear}(C) \Rightarrow \text{On}(A, Fl)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B) \lor \text{Clear}(A)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B)$</td>
<td>$B$</td>
</tr>
<tr>
<td>$\text{Clear}(C)$</td>
<td>$C$</td>
</tr>
</tbody>
</table>

## Knowledge

Does the following knowledge base (set of formulae) have a model?

<table>
<thead>
<tr>
<th>Formula</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{On}(A, Fl) \Rightarrow \text{Clear}(B)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B) \land \text{Clear}(C) \Rightarrow \text{On}(A, Fl)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B) \lor \text{Clear}(A)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(B)$</td>
<td></td>
</tr>
<tr>
<td>$\text{Clear}(C)$</td>
<td></td>
</tr>
</tbody>
</table>

The above figure illustrates the three possible blocks-world situations.
An example

$$(\forall x) \ [\text{On}(x,C) \implies \neg \text{Clear}(C)]$$

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15.2**
Three Blocks-World Situations

Closed Formulas

- Recall: A wff is closed if it contains no free occurrences of any variable.
- We will mostly restrict ourselves to closed formulas.
- For formulas with free variables, close the formula by universally quantifying over all its free variables.
Validity (Tautologies)

- **Definition.** A predicate logic formula is **satisfiable** if there is an interpretation and there is a valuation that satisfies the formula (i.e., in which the formula returns T).

- **Definition.** A predicate logic formula is **logically valid (tautology)** if it is true in every interpretation.
  - It must be satisfied by every valuation in every interpretation.

- **Definition.** A wff, A, of predicate logic is a **contradiction** if it is false in every interpretation.
  - It must be false in every valuation in every interpretation.

Satisfiability, Tautologies, Contradictions

- A closed predicate logic formula, is **satisfiable** if there is an interpretation I in which the formula returns true.

- A closed predicate logic formula, A, is a **tautology** if it is true in every interpretation.
  \[ \models A \]

- A closed predicate logic formula is a **contradiction** if it is false in every interpretation.
Tautologies

- How can we check if a formula is a tautology?
- If the domain is finite, then we can try all the possible interpretations (all the possible functions and predicates).
- But if the domain is infinite? Intuitively, this is why a computer cannot be programmed to determine if an arbitrary formula in predicate logic is a tautology (for all tautologies).
- Our only alternative is proof procedures!
- Therefore the soundness and completeness of our proof procedures is very important!

Semantic Entailment

Semantic entailment has the same meaning as it did for propositional logic.

$$\phi_1, \phi_2, \phi_3 \vdash \psi$$

means that if $\nu(\phi_1) = T$ and $\nu(\phi_2) = T$ and $\nu(\phi_3) = T$
then $\nu(\psi) = T$, which is equivalent to saying

$$(\phi_1 \land \phi_2 \land \phi_3) \Rightarrow \psi$$

is a tautology, i.e.,

$$(\phi_1, \phi_2, \phi_3 \vdash \psi) \equiv ((\phi_1 \land \phi_2 \land \phi_3) \Rightarrow \psi)$$
An Axiomatic System for Predicate Logic

FO_AL: An extension of the axiomatic system for propositional logic. Use only: \( \Rightarrow, \neg, \forall \)

\[
A \Rightarrow (B \Rightarrow A)
\]

\[
(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))
\]

\[
(\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A)
\]

\[
\forall x . A(x) \Rightarrow A(t), \text{ where } t \text{ is free for } x \text{ in } A
\]

\[
\forall x . (A \Rightarrow B) \Rightarrow (A \Rightarrow (\forall x . B)), \text{ where } A \text{ contains no free occurrences of } x
\]

---

FO_AL Rules of Inference

Two rules of inference:

- (modus ponens - MP) From \( A \) and \( A \Rightarrow B \), \( B \) can be derived, where \( A \) and \( B \) are any well-formed formulas.

- (generalization) From \( A \), \( \forall x . A \) can be derived, where \( A \) is any well-formed formula and \( x \) is any variable.
Soundness and Completeness of FO_AL

- FO_AL is sound and complete.
- Completeness was proven by Kurt Gödel in 1929 in his doctoral dissertation.
- Predicate logic is not decidable

Deduction Theorem

- **Theorem.** If \( H \cup \{ A \} \vdash B \) by a deduction containing no application of generalization to a variable that occurs free in \( A \), then \( H \vdash A \Rightarrow B \)
- **Corollary.** If \( A \) is closed and if \( H \cup \{ A \} \vdash B \) then \( H \vdash (A \Rightarrow B) \)
Proof by Refutation

- A closed formula is a tautology (valid) iff its negation is a contradiction.
- In other words, a closed formula is valid iff its negation is not satisfiable.
- To prove \( \{P_1, \ldots, P_n\} \vdash S \) is equivalent to prove \( \{P_1, \ldots, P_n, \neg S\} \vdash \text{false} \).
- To prove \( \{P_1, \ldots, P_n\} \vdash S \) becomes to check if there is an interpretation for \( \{P_1, \ldots, P_n, \neg S\} \).

How many interpretations are there?

Counterexamples

- How can we show a formula is not a tautology?
- Provide a counterexample. A counterexample for a closed formula is an interpretation in which the formula does not have the truth value T.
Example

Prove \( \forall x. \forall y. A \vdash \forall y. \forall x. A \)

1. \( \forall x. \forall y. A \) premise
2. \( \forall x. \forall y. A \Rightarrow \forall y. A \) Ax4
3. \( \forall y. A \) MP 1, 2
4. \( \forall y. A \Rightarrow A \) Ax4
5. \( A \) MP 3, 4
6. \( \forall x. A \) Gen of 5
7. \( \forall y. \forall x. A \) Gen of 6

Workout: Counterexamples

Show that \((\forall x. P(x) \lor Q(x)) \iff ((\forall x. P(x)) \lor (\forall x. Q(x)))\) is not a tautology by constructing a model that makes the formula false.
What does ‘first-order’ mean?

- We can only quantify over variables.
- In higher-order logics, we can quantify over functions, and predicates.
  - For example, in second-order logic, we can express the induction principle:
    \[ \forall P. (P(0) \land (\forall n. P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n. P(n)) \]
- Propositional logic can also be thought of as zero-order.

A rough timeline in ATP ... (1/3)

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>450 B.C.</td>
<td>Stoics: propositional logic (PL), inference (maybe)</td>
</tr>
<tr>
<td>322 B.C.</td>
<td>Aristotle: 'syllogism' (inference rules), quantifiers</td>
</tr>
<tr>
<td>1565</td>
<td>Cardano: probability theory (PL + uncertainty)</td>
</tr>
<tr>
<td>1646</td>
<td>Leibniz: research for a general decision procedure to check the validity of formulas</td>
</tr>
<tr>
<td>-1716</td>
<td>Leibniz - In 1716, Leibniz proposed a general decision procedure to check the validity of formulas.</td>
</tr>
<tr>
<td>1847</td>
<td>Boole: PL (again)</td>
</tr>
<tr>
<td>1879</td>
<td>Frege: first-order logic (FOL)</td>
</tr>
<tr>
<td>1889</td>
<td>Peano: 9 axioms for natural numbers</td>
</tr>
</tbody>
</table>
A rough timeline in ATP ...

1920s Hilbert’s program

1922 Wittgenstein proof by truth tables

1929 Gödel completeness theorem of FOL

1930 Herbrand a proof procedure for FOL based on propositionalization

1931 Gödel incompleteness theorems for the consistency of Peano axioms

1936 Gentzen a proof for the consistency of Peano axioms in set theory

1936 Church, Turing undecidability of FOL

1936 Gentzen a method to prove the consistency of Peano axioms with type theory

1954 Davis First machine-generated proof

1955 Beth, Hintikka Semantic Tableaus

1957 Newell, Simon First machine-generated proof in Logic Calculus

1957 Kangar, Prawitz Lazy substitution by free (dummy) Vars

1958 Prawitz First prover for FOL

1959 Gilmore More provers

1960 Davis, Putnam, Longman Davis-Putnam Procedure

1963 Robinson Unification, resolution

A rough timeline in ATP ...

• To formalize all existing theories to a finite, complete, and consistent set of axioms.
• Decision procedures for all mathematical theories
• 23 open problems.
Kurt Gödel
1906 - 1978
• Born an Austro-Hungarian
• 12 → Czech
• refuse to learn Czech
• 23 → Austrian
• established the completeness of 1st-order logic in his Ph.D. thesis
• 25, established the incompleteness of \( \mathbb{N} \)
• 32 → German
• 34 → joined Princeton
• 42 → American
• Einstein, "his work no longer meant much, that he came to the Institute merely … to have the privilege of walking home with Gödel."
• On his citizen exam, …
• proved a paradoxical solution to the general relativity
• Permanent position, Princeton, 1946
• 1st Albert Einstein Award, 1951
• Full professor, 1953
• National Science Medal, 1974
• Emeritus professor, 1976

I knew the general relativity was wrong.

• ate only his wife’s cooking.
• 1977, his wife was ill and could not cook.
• Jan. 1978, died of mal-nutrition.

American is in danger of dictatorship because I can prove the contradiction in American constitution.

I thought someone was to poison him.

2007/04/03 stopped here.
Predicate Logic: Natural Deduction

Extend the set of rules we used for propositional logic with ones to handle quantifiers.

**Universal Quantification**

\[
\forall x. P \\
\frac{P[t/x]}{\forall x. P} \quad \forall e
\]

\[
\begin{array}{c}
\forall x. P \\
\vdots \\
\therefore P[x_0/x] \\
\forall x. P
\end{array} \quad \forall i
\]

\(x_0\) must be arbitrary, meaning it doesn’t appear outside the subproof. \(t\) must be free for \(x\) in \(P\).

**Existential Quantification**

\[
\exists x. P \\
\frac{P[t/x]}{\exists x. P} \quad \exists i
\]

\[
\begin{array}{c}
x_0, P[x_0/x] \\
\vdots \\
\therefore Q \\
\exists e
\end{array}
\]

Informally: If we know that the predicate is true for some value, and using an arbitrary variable, we derive that a formula holds, then we can conclude that the formula holds.

\(x_0\) must be arbitrary. \(t\) must be free for \(x\) in \(P\).
Example

Show $\forall x. P(x) \Rightarrow Q(x)$, $\forall x. P(x) \vdash \forall x. Q(x)$

1. $\forall x. P(x) \Rightarrow Q(x)$  premise
2. $\forall x. P(x)$  premise
3. $x_0$
4. $P(x_0) \Rightarrow Q(x_0)$  $\forall e$ 1
5. $P(x_0)$  $\forall e$ 2
6. $Q(x_0)$  $\Rightarrow e$ 4, 5
7. $\forall x. Q(x)$  $\forall i$ 3 – 6

Workout

- Show $P(a), \forall x. P(x) \Rightarrow \neg Q(x) \vdash \neg Q(a)$
- Show $\neg \forall x. P(x) \vdash \exists x. \neg P(x)$
Proof by Refutation

- To prove \( \{P_1, \ldots, P_n\} \models S \) is equivalent to prove that there is no interpretation for \( \{P_1, \ldots, P_n, \neg S\} \).
- But there are infinitely many interpretations!
- Can we limit the range of interpretations?
- Yes, Herbrand interpretations!

---

Herbrand’s theorem
- Herbrand universe of a formula \( S \)

- Let \( H_0 \) be the set of constants appearing in \( S \).
  - If no constant appears in \( S \), then \( H_0 \) is to consist of a single constant, \( H_0 = \{a\} \).
- For \( i = 0, 1, 2, \ldots \)
  
  \[ H_{i+1} = H_i \cup \{f^n(t_1, \ldots, t_n) \mid f \text{ is an } n\text{-place function in } S; \ t_1, \ldots, t_n \in H_i \} \]
- \( H_i \) is called the \( i \)-level constant set of \( S \).
- \( H_\infty \) is the Herbrand universe of \( S \).
Herbrand’s theorem
- Herbrand universe of a formula $S$

Example 1: $S=\{P(a), \sim P(x) \lor P(f(x))\}$
- $H_0=\{a\}$
- $H_1=\{a, f(a)\}$
- $H_2=\{a, f(a), f(f(a))\}$
- ...
- $H_\infty=\{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$

Herbrand’s theorem
- Herbrand universe of a formula $S$

Example 2: $S=\{P(x) \lor Q(x), R(z), T(y) \lor \sim W(y)\}$
- There is no constant in $S$, so we let $H_0=\{a\}$
- There is no function symbol in $S$, hence $H=H_0=H_1=\ldots=\{a\}$

Example 3: $S=\{P(f(x), a, g(y), b)\}$
- $H_0=\{a, b\}$
- $H_1=\{a, b, f(a), f(b), g(a), g(b)\}$
- $H_2=\{a, b, f(a), f(b), g(a), g(b), f(f(a)), f(f(b)), f(g(a)), f(g(b)), g(f(a)), g(f(b)), g(g(a)), g(g(b))\}$
- ...

...
Herbrand’s theorem  
- Herbrand universe of a formula $S$

**Expression**
- a term, a set of terms, an atom, a set of atoms, a literal, a clause, or a set of clauses.

**Ground expressions**
- expressions without variables.

It is possible to use a ground term, a ground atom, a ground literal, and a ground clause – this means that no variable occurs in respective expressions.

**Subexpression** of an expression $E$
- an expression that occurs in $E$.

---

Herbrand’s theorem  
- Herbrand base of a formula $S$

- Ground atoms $P^n(t_1,\ldots,t_n)$
  - $P^n$ is an $n$-place predicate occurring in $S$,
  - $t_1,\ldots,t_n \in H_\infty$

- Herbrand base of $S$ (atom set)
  - the set of all ground atoms of $S$

- Ground instance of $S$
  - obtained by replacing variables in $S$ by members of the Herbrand universe of $S$. 

---

57

---

58
Herbrand’s theorem
- Herbrand universe & base of a formula S

Example
- $S = \{P(x), Q(f(y)) \lor R(y)\}$
- $C = P(x)$ is a clause in $S$
- $H = \{a, f(a), f(f(a)), \ldots\}$ is the Herbrand universe of $S$
- $P(a), Q(f(a)), Q(a), R(a), R(f(f(a)))$, and $P(f(f(a)))$ are ground atoms of $C$.

Workout

$\{P(x), Q(g(x,y),a) \lor R(f(x))\}$
- please construct the set of ground terms
- please construct the set of ground atoms
Herbrand’s theorem
- Herbrand interpretation of a formula S

- S, a set of clauses.
  - i.e., a conjunction of the clauses
- H, the Herbrand universe of S and
- H-interpretation \( \mathcal{I} \) of S
  - \( \mathcal{I} \) maps all constants in S to themselves.
  - For all n-place function symbol \( f \) and \( h_1,\ldots,h_n \) elements of \( H \),
    \[ \mathcal{I}(f(h_1,\ldots,h_n)) = f(h_1,\ldots,h_n) \]

- Herbrand’s theorem
- Herbrand interpretation of a formula S

- There is no restriction on the assignment to each n-place predicate symbol in S.
- Let \( A=\{A_1,A_2,\ldots,A_n,\ldots\} \) be the atom set of S.
- An H-interpretation \( I \) can be conveniently represented as a subset of \( A \).
  - If \( A_j \in I \), then \( A_j \) is assigned “true”,
  - otherwise \( A_j \) is assigned “false”.
Herbrand’s theorem
- **Herbrand interpretation of a formula S**

Example: $S = \{P(x) \lor Q(x), R(f(y))\}$

- The Herbrand universe of $S$ is $H = \{a, f(a), f(f(a)), \ldots\}$.
- Predicate symbols: $P, Q, R$.
- The atom set of $S$: $A = \{P(a), Q(a), R(a), P(f(a)), Q(f(a)), R(f(a)), \ldots\}$.
- Some $H$-interpretations for $S$:
  - $I_1 = \{P(a), Q(a), R(a), P(f(a)), Q(f(a)), R(f(a)), \ldots\}$
  - $I_2 = \emptyset$
  - $I_3 = \{P(a), Q(a), P(f(a)), Q(f(a)), \ldots\}$

---

Herbrand’s theorem
- **Herbrand interpretation of a formula S**

- An interpretation of a set $S$ of clauses does not necessarily have to be defined over the Herbrand universe of $S$.
- Thus an interpretation may not be an $H$-interpretation.

Example:

- $S = \{P(x), Q(y, f(y, a))\}$
- $D = \{1, 2\}$
Herbrand’s theorem
- Herbrand interpretation of a formula $S$

- But Herbrand is conceptually general enough.

Example (cont.) $S=\{P(x),Q(y,f(y,a))\}$
- $D=\{1,2\}$
- an interpretation of $S$:

$$
\begin{array}{c|cccc}
  a & f(1,1) & f(1,2) & f(2,1) & f(2,2) \\
  \hline
  2 & 1 & 2 & 2 & 1
\end{array}
$$

$$
\begin{array}{cccccccc}
  & P(1) & P(2) & Q(1,1) & Q(1,2) & Q(2,1) & Q(2,2) \\
  \hline
  T & F & F & T & F & T
\end{array}
$$

Herbrand’s theorem
- Herbrand interpretation of a formula $S$

- But Herbrand is conceptually general enough.

Example (cont.) – we can define an $H$-interpretation $I^*$ corresponding to $I$.

First we find the atom set of $S$
- $A=\{P(a),Q(a,a),P(f(a,a)),Q(a,f(a,a)),Q(f(a,a),a),Q(f(a,a),f(a,a))\}$

Next we evaluate each member of $A$ by using the given table

$$
\begin{array}{c|cccc}
  a & f(1,1) & f(1,2) & f(2,1) & f(2,2) \\
  \hline
  2 & 1 & 2 & 2 & 1
\end{array}
$$

$$
\begin{array}{cccccccc}
  & P(1) & P(2) & Q(1,1) & Q(1,2) & Q(2,1) & Q(2,2) \\
  \hline
  T & F & F & T & F & T
\end{array}
$$

Then $I^*=\{Q(a,a),P(f(a,a)),Q(f(a,a),a),\ldots\}$. 
Herbrand’s theorem
- *Herbrand interpretation of a formula S*

- If there is no constant in $S$, the element $a$ used to initiate the Herbrand universe of $S$ can be mapped into any element of the domain $D$.
- If there is more than one element in $D$, then there is more than one $H$-interpretation corresponding to $I$.

Example: $S=\{P(x), Q(y, f(y, z))\}$, $D=\{1, 2\}$

<table>
<thead>
<tr>
<th>$f(1,1)$</th>
<th>$f(1,2)$</th>
<th>$f(2,1)$</th>
<th>$f(2,2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$P(1)$</th>
<th>$P(2)$</th>
<th>$Q(1,1)$</th>
<th>$Q(1,2)$</th>
<th>$Q(2,1)$</th>
<th>$Q(2,2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

- Two $H$-interpretations corresponding to $I$ are:
  - $I^*=\{Q(a,a), P(f(a,a)), Q(f(a,a), a)\ldots\}$ if $a=2$,
  - $I^*=\{P(a), P(f(a,a)), \ldots\}$ if $a=1$. 

Herbrand’s theorem

-Herbrand interpretation of a formula $S$

Definition: Given an interpretation $I$ over a domain $D$, an $H$-interpretation $I^*$ corresponding to $I$ is an $H$-interpretation that satisfies the condition:

- Let $h_1, \ldots, h_n$ be elements of $H$ (the Herbrand universe of $S$).
- Let every $h_i$ be mapped to some $d_i$ in $D$.
- If $P(d_1, \ldots, d_n)$ is assigned $T$ ($F$) by $I$, then $P(h_1, \ldots, h_n)$ is also assigned $T$ ($F$) in $I^*$.

Lemma: If an interpretation $I$ over some domain $D$ satisfies a set $S$ of clauses, then any $H$-interpretation $I^*$ corresponding to $I$ also satisfies $S$.

Herbrand’s theorem

A set $S$ of clauses is unsatisfiable if and only if $S$ is false under all the $H$-interpretations of $S$.

- We need consider only $H$-interpretations for checking whether or not a set of clauses is unsatisfiable.
- Thus, whenever the term “interpretation” is used, a $H$-interpretation is meant.
Herbrand’s theorem

Let $\emptyset$ denote an empty set. Then:

- A ground instance $C'$ of a clause $C$ is satisfied by an interpretation $I$ if and only if there is a ground literal $L'$ in $C'$ such that $L'$ is also in $I$, i.e. $C \cap I \neq \emptyset$.
- A clause $C$ is satisfied by an interpretation $I$ if and only if every ground instance of $C$ is satisfied by $I$.
- A clause $C$ is falsified by an interpretation $I$ if and only if there is at least one ground instance $C'$ of $C$ such that $C'$ is not satisfied by $I$.
- A set $S$ of clauses is unsatisfiable if and only if for every interpretation $I$ there is at least one ground instance $C'$ of some clause $C$ in $S$ such that $C'$ is not satisfied by $I$.

Example: Consider the clause $C = P(x) \lor Q(f(x))$. Let $I_1$, $I_2$, and $I_3$ be defined as follows:
- $I_1 = \emptyset$
- $I_2 = \{P(a), Q(a), P(f(a)), Q(f(a)), P(f(f(a))), Q(f(f(a))), \ldots\}$
- $I_3 = \{P(a), P(f(a)), P(f(f(a))), \ldots\}$
$C$ is satisfied by $I_1$ and $I_2$, but falsified by $I_3$.

Example: $S = \{P(x), \neg P(a)\}$.
The only two $H$-interpretations are:
- $I_1 = \{P(a)\}$
- $I_2 = \emptyset$.
$S$ is falsified by both $H$-interpretations and therefore is unsatisfiable.
Resolution Principle
- Clausal Forms

Clauses are universally quantified disjunctions of literals;
all variables in a clause are universally quantified

\[(\forall x_1, \ldots, x_n)(l_1 \lor \ldots \lor l_n)\]

written as
\[l_1 \lor \ldots \lor l_n\]
or
\[\{l_1, \ldots, l_n\}\]

Examples:

\[
\{\text{Nat}(s(A)), \neg\text{Nat}(A)\}
\]
gives
\[
\{\text{Nat}(A)\}
\]
gives
\[
\{\text{Nat}(s(A))\}
\]

\[
\{\text{Nat}(s(s(x))), \neg\text{Nat}(s(x))\}
\]
gives
\[
\{\text{Nat}(s(A))\}
\]
gives
\[
\{\text{Nat}(s(s(A)))\}
\]

We need to be able to work with variables!

Unification of two expressions/literals
Resolution Principle - Terms and instances

- Consider following atoms

\[ P(x,f(y),B) \]
\[ P(z,f(w),B) \text{ alphabetic variant} \]
\[ P(x,f(A),B) \text{ instance} \]
\[ P(g(z),f(A),B) \text{ instance} \]
\[ P(C,f(A),A) \text{ not an instance} \]

- **Ground** expressions do not contain any variables

Resolution Principle - Substitution

A substitution \( s = \{ t_1/v_1, \ldots, t_n/v_n \} \) substitutes variables \( v_i \) for terms \( t_i \) (\( t_i \) does NOT contain \( v_i \))

Applying a substitution \( s \) to an expression \( \omega \)
yields the expression \( \omega s \) which is \( \omega \)
with all occurrences of \( v_i \) replaced by \( t_i \)

\[ P(x,f(y),B) \]
\[ P(z,f(w),B) \quad s = \{ z/x, w/y \} \]
\[ P(x,f(A),B) \quad s = \{ A/y \} \]
\[ P(g(z),f(A),B) \quad s = \{ g(z)/x, A/y \} \]
\[ P(C,f(A),A) \quad \text{no substitution} ! \]
Workout

Calculate the substitutions for the resolution of the two clauses and the result clauses after the substitutions.

- \( \neg P(x), P(f(a)) \lor Q(f(y), g(a,b)) \)
- \( \neg P(g(x,a)), P(y) \lor Q(f(y), g(a,b)) \)
- \( \neg P(g(x,f(a))), P(g(b,y)) \lor Q(f(y), g(a,b)) \)
- \( \neg P(g(f(x),x)), P(g(y,f(y))) \lor Q(f(y), g(a,b)) \)

Resolution Principle

- Composing substitutions

Composing substitutions \( s_1 \) and \( s_2 \) gives \( s_1 \cdot s_2 \) which is that substitution obtained by first applying \( s_2 \) to the terms in \( s_1 \) and adding remaining term/vars pairs to \( s_1 \).

\[
\theta = \{ g(x,y) / z \} \{ A/x, B/y, C/w, D/z \} = \\
\{ g(A,B) / z, A/x, B/y, C/w \}
\]

Apply to

\[
P(x,y,z) \theta \\
gives \\
P(A,B,g(A,B))
\]
Resolution Principle
- Properties of substitutions

\[(\omega s_1)s_2 = \omega(s_1s_2)\]
\[(s_1s_2)s_3 = s_1(s_2s_3)\]  associativity

\[s_1s_2 \neq s_2s_1\]  not commutative

Resolution Principle
- Unification

- Unifying a set of expressions \(\{w_i\}\)
  - Find substitution \(s\) such that \(w_is = w_js\) for all \(i, j\)
  - Example

\[
\{ P(x, f(y), B), P(x, f(B), B) \}
\]

\[
s = \{B/y, A/x\}\]  not the simplest unifier

\[
s = \{B/y\}\]  most general unifier (mgu)

- The most general unifier, the mgu, \(g\) of \(\{w_i\}\) has the property that if \(s\) is any unifier of \(\{w_i\}\) then there exists a substitution \(s'\) such that \(\{w_i\}s = \{w_i\}gs'\)

- The common instance produced is unique up to alphabetic variants (variable renaming)

- Usually we assume there is no common variables in the two atoms.
Workout

\[ P(B, f(x), g(A)) \] and \[ P(y, z, f(w)) \]
- construct an mgu
- construct a unifier that is not the most general.

---

Workout

Determine if each of the following sets is unifiable. If yes, construct an mgu.

- \{Q(a), Q(b)\}
- \{Q(a, x), Q(a, a)\}
- \{Q(a, x, f(x)), Q(a, y, y)\}
- \{Q(x, y, z), Q(u, h(v, v), u)\}
- \{P(x_1, g(x_1), x_2, h(x_1, x_2), x_3, k(x_1, x_2, x_3)), P(y_1, y_2, e(y_2), y_3, f(y_2, y_3), y_4)\}
Resolution Principle
- Disagreement set in unification

The disagreement set of a set of expressions \( \{ w_i \} \) is the set of subterms \( \{ t_i \} \) of \( \{ w_i \} \) at the first position in \( \{ w_i \} \) for which the \( \{ w_i \} \) disagree.

\[
\begin{align*}
\{ P(x,A,f(y)) , P(w,B,z) \} & \text{ gives } \{ x,w \} \\
\{ P(x,A,f(y)) , P(x,B,z) \} & \text{ gives } \{ A,B \} \\
\{ P(x,y,f(y)) , P(x,B,z) \} & \text{ gives } \{ y,B \}
\end{align*}
\]

Resolution Principle
- Unification algorithm

Unify(\( Terms \))
Initialize \( k \leftarrow 0 \);
Initialize \( T_k = Terms \);
Initialize \( \sigma_k = \{ \} \);
* If \( T_k \) is a singleton, then output \( \sigma_k \). Otherwise, continue.
Let \( D_k \) be the disagreement set of \( T_k \).
If there exists a var \( v_k \) and a term \( t_k \) in \( D_k \) such that \( v_k \) does not occur in \( t_k \), continue. Otherwise, exit with failure.
\[
\begin{align*}
\sigma_{k+1} & \leftarrow \sigma_k \{ t_k / v_k \} \\
T_{k+1} & \leftarrow T_k \{ t_k / v_k \} \\
k & \leftarrow k + 1
\end{align*}
\]
Goto *
Predicate calculus Resolution

John Allan Robinson (1965)

Let $C_1$ and $C_2$ be two clauses with literals $l_1 \in C_1$ and $\neg l_2 \in C_2$ such that $C_1$ and $C_2$ do not contain common variables, and $mgu(l_1, l_2) = \theta$
then $C = \{ \{ C_1 - \{l_1\} \} \cup \{ C_2 - \{-l_2\} \} \} \theta$
is a resolvent of $C_1$ and $C_2$

Predicate calculus Resolution

John Allan Robinson (1965)

Given

$C: \lor \lor \ldots \lor \lor$
$C: \lor \lor \ldots \lor \lor$
$\theta = mgu(l_1, k_1)$
the resolvent is

$l_2 \theta \lor \ldots \lor l_m \theta \lor k_2 \theta \lor \ldots \lor k_n \theta$
Resolution Principle - Example

Why can we do this?
Why we think the variables in 2 clauses are irrelevant?

\[
P(x) \lor Q(f(x)) \text{ and } R(g(x)) \lor \neg Q(f(A))
\]
Standardizing the variables apart

\[
P(x) \lor Q(f(x)) \text{ and } R(g(y)) \lor \neg Q(f(A))
\]
Substitution \( \theta = (A/x) \)
Resolvent \( P(A) \lor R(g(y)) \)

\[
P(x) \lor Q(x,y) \text{ and } \neg P(A) \lor \neg R(B,z)
\]
Standardizing the variables apart

Substitution \( \theta = (A/x) \)
Resolvent \( Q(A,y) \lor \neg R(B,z) \)

Workout

Find all the possible resolvents (if any) of the following pairs of clauses.

- \( \neg P(x) \lor Q(x,b) \),

- ...
Workout

Find all the possible resolvents (if any) of the following pairs of clauses.

- \( \neg P(x) \lor Q(x,b), P(a) \lor Q(a,b) \)
- \( \neg P(x) \lor Q(x,x), \neg Q(a,f(a)) \)
- \( \neg P(x,y,u) \lor \neg P(y,z,v) \lor \neg P(x,v,w) \lor P(u,z,w), \)
  \( P(g(x,y),x,y) \)
- \( \neg P(v,z,v) \lor P(w,z,w), P(w,h(x,x),w) \)

Resolution Principle

- A stronger version of resolution

Use more than one literal per clause

\{ P(u), P(v) \} and \{ \neg P(x), \neg P(y) \} 

do not resolve to empty clause.

However, ground instances

\{ P(A) \} and \{ \neg P(A) \} resolve to empty clause
Resolution Principle
- Factors

Let $C_1$ be a clause such that there exists a substitution $\theta$ that is a mgu of a set of literals in $C_1$. Then $C_1\theta$ is a factor of $C_1$

Each clause is a factor of itself.
Also, $\{P(f(y)),R(f(y),y)\}$ is a factor of $\{P(x),P(f(y)),R(x,y)\}$ with $\theta = \{f(y)/x\}$

Resolution Principle
- Example of refutation

1. $\{F(\text{Art},\text{Jon})\}$ \(\Delta\)
2. $\{F(\text{Bob},\text{Kim})\}$ \(\Delta\)
3. $\{\neg F(x,y), P(x,y)\}$ \(\Delta\)
4. $\{\neg P(\text{Art},\text{Jon})\}$ \(\Gamma\)
5. $\{P(\text{Art},\text{Jon})\}$ \(1, 3\)
6. $\{P(\text{Bob},\text{Kim})\}$ \(2, 3\)
7. $\{\neg F(\text{Art},\text{Jon})\}$ \(3, 4\)
8. $\{}$ \(4, 5\)
9. $\{}$ \(1, 7\)
Resolution Principle
- Example

**Hypotheses**

\[ \forall x \ (\text{dog}(x) \Rightarrow \text{animal}(x)) \]
\[ \text{dog}(\text{fido}) \]
\[ \forall y \ (\text{animal}(y) \Rightarrow \text{die}(y)) \]

**Clausal Form**

\[ \neg\text{dog}(x) \lor \text{animal}(x) \]
\[ \text{dog}(\text{fido}) \]
\[ \neg\text{animal}(y) \lor \text{die}(y) \]

**Conclusion**

\[ \text{die}(\text{fido}) \]

Negate the goal

\[ \neg\text{die}(\text{fido}) \]
Workout (resolution)
- Proof with resolution principle

Hypotheses:
- $P(m(x), x) \lor Q(m(x))$
- $\neg P(y, z) \lor R(y)$
- $\neg Q(m(f(x,y))) \lor \neg T(x, g(y))$
- $S(a) \lor T(f(a), g(x))$
- $\neg R(m(y))$
- $\neg S(x) \lor W(x, f(x, y))$

Conclusion
$W(a, y)$

Resolution

Properties
- Resolution is sound
- Incomplete

Given $P(A)$
Infer $\{P(A), P(B)\}$

But fortunately it is refutation complete
- If KB is unsatisfiable then KB $\vdash \Box$
Resolution Principle
- Refutation Completeness

To decide whether a formula $KB \models w$, do

1. Convert $KB$ to clausal form $KB'$
2. Convert $\neg w$ to clausal form $\neg w'$
3. Combine $\neg w'$ and $KB'$ to give $\Delta$
4. Iteratively apply resolution to $\Delta$ and add the results back to $\Delta$ until either no more resolvents can be added, or until the empty clause is produced.

Resolution Principle
- Converting to clausal form (1/2)

To convert a formula $KB$ into clausal form

1. Eliminate implication signs*

   $(p \Rightarrow q)$ becomes $(\neg p \lor q)$

2. Reduce scope of negation signs*

   $(p \land q)$ becomes $(\neg p \lor \neg q)$

3. Standardize variables

   $(\forall x) [\neg P(x) \lor (\exists x) Q(x)]$ becomes $(\forall x) [\neg P(x) \lor (\exists y) Q(y)]$

4. Eliminate existential quantifiers using Skolemization

* Same as in prop. logic
Resolution Principle
- Converting to clausal form (2/2)

5. Convert to prenex form
   - Move all universal quantifiers to the front

6. Put the matrix in conjunctive normal form*
   - Use distribution rule

7. Eliminate universal quantifiers

8. Eliminate conjunction symbol *

9. Rename variables so that no variable occurs in more than one clause.

Resolution Principle
- Skolemization

Consider $(\forall x)[(\exists y)\text{Height}(x,y)]$

The $y$ depends on the $x$

Define this dependence explicitly using a skolem function $h(x)$

Formula becomes $(\forall x)[\text{Height}(x,h(x))]$

General rule is that each occurrence of an existentially quantified variable is replaced by a skolem function whose arguments are those universally quantified variables whose scopes includes the scope of the existentially quantified one

Skolem functions do not yet occur elsewhere!

Resulting formula is not logically equivalent!
Resolution Principle
- Examples of Skolemization

Example of conversion to clausal form

\[
(\forall x) [ (\exists y) \ F(x,y) ] \quad \text{gives} \quad (\forall x) \ F(x,h(x))
\]

but

\[
(\exists y) [ (\forall x) \ F(x,y) ] \quad \text{gives} \quad [ (\forall x) \ F(x,sk) ] \quad \text{skolem constant Not logically equivalent!}
\]

A well formed formula and its Skolem form are not logically equivalent.

However, a set of formulae is (un)satisfiable if and only if its skolem form is (un)satisfiable.
Workout

Convert the following formula to clausal form.

- $\exists x (P(x) \land \forall y ((\exists z. Q(x,y,s(z))) \rightarrow (Q(x,s(y),x) \land R(y))))$
- $\forall x \forall y (S(x,y,z) \rightarrow \exists z (S(x,z) \land S(z,x)))$

Resolution Principle
- Example of refutation by resolution

- all packages in room 27 are smaller than any of those in 28
- Prove $\neg \neg \neg \neg \neg \neg$

1. $P(x) \lor P(y) \lor I(x,27) \lor I(y,28) \lor S(x,y)$
2. $P(A)$
3. $P(B)$
4. $I(A,27) \lor I(A,28)$
5. $I(B,27)$
6. $\neg S(B,A)$
Prove $I(A,27)$
Resolution Principle
- Search Strategies

- Ordering strategies
  - In what order to perform resolution?
  - Breadth-first, depth-first, iterative deepening?
  - Unit-preference strategy:
    - Prefer those resolution steps in which at least one clause is a unit clause (containing a single literal)

- Refinement strategies
  - Unit resolution: allow only resolution with unit clauses

Resolution Principle
- Input Resolution

- at least one of the clauses being resolved is a member of the original set of clauses
- Input resolution is complete for Horn-clauses but incomplete in general
- E.g. \{P, Q\}, \{\neg P, Q\}, \{P, \neg Q\}, \{\neg P, \neg Q\}

- One of the parents of the empty clause should belong to original set of clauses
Workout

Use input resolution to prove the theorem in page workout(resolution)!

Resolution Principle
- Linear Resolution

- Linear resolvent is one in which at least one of the parents is either
  - an initial clause or
  - the resolvent of the previous resolution step.

- Refutation complete
- Many other resolution strategies exist
Use linear resolution to prove the theorem in
page workout(resolution)!

Resolution Principle
- Set of support
  - Ancestor : c2 is a descendant of c1 iff c2 is a
    resolvent of c1 (and another clause) or if c2 is a
    resolvent of a descendant of c1 (and another
    clause); c1 is an ancestor of c2
  - Set of support : the set of clauses coming from
    the negation of the theorem (to be proven) and
    their descendants
  - Set of support strategy : require that at least one
    of the clauses in each resolution step belongs to
    the set of support
workout

Use set of support to prove the theorem in page workout(resolution)!

Resolution Principle
- Answer extraction

Suppose we wish to prove whether KB |= (∃w)f(w)
We are probably interested in knowing the w for which f(w) holds.
Add Ans(w) literal to each clause coming from the negation of the theorem to be proven; stop resolution process when there is a clause containing only Ans literal
Resolution Principle
- Example of answer extraction

1. \( \neg P(x) \lor \neg P(y) \lor \neg I(x,27) \lor \neg I(y,28) \lor S(x,y) \)
   all packages in room 27 are smaller than any of those in 28

2. \( P(A) \)

3. \( P(B) \)

4. \( I(A,27) \lor I(A,28) \)

5. \( I(B,27) \)

6. \( \neg S(B,A) \)

Prove \( (\exists u) I(A,u) \), i.e. in which room is A?

---

Workout

- Use answer extraction to prove the theorem in page workout(resolution)!
Theory of Equality

- Herbrand Theorem does not apply to FOL with equality.
- So far we’ve looked at predicate logic from the point of view of what is true in all interpretations.
  - This is very open-ended.
- Sometimes we want to assume at least something about our interpretation to enrich the theory in what we can express and prove.
- The meaning of equality is something that is common to all interpretations.
  - Its interpretation is that of equivalence in the domain.
  - If we add = as a predicate with special meaning in predicate logic, we can also add rules to our various proof procedures.
- Normal models are models in which the symbol = is interpreted as designating the equality relation.

Theory of Equality
- An Axiomatic System with Equality

To the previous axioms and rules of inference, we add:

EAx1 $\forall x. x = x$
EAx2 $\forall x. \forall y. x = y \Rightarrow (A(x, x) \Rightarrow A(x, y))$
EAx3 $\forall x. \forall y. x = y \Rightarrow f(x) = f(y)$
Theory of Equality
- Natural Deduction Rules for Equality

Reflexivity

\[ t = t \]

This inference rule is called an \textit{axiom}, because it has no premises.

---

Substitution

\[ t_1 = t_2 \quad P[t_2/x] \]

\[ P[t_1/x] = e \]

\[ t_1 = t_2 \quad P[t_1/x] \]

\[ P[t_2/x] = e \]

where \( t_1 \) and \( t_2 \) are free in \( x \) in \( P \).
Theory of Equality
- Substitution

- Recall: Given a variable $x$, a term $t$ and a formula $P$, we define $P[t / x]$ to be the formula obtained by replacing ALL free occurrence of variable $x$ in $P$ with $t$.

- But with equality, we sometimes don’t want to substitute for all occurrences of a variable.

- When we write $P[t / x]$ above the line, we get to choose what $P$ is and therefore can choose the occurrences of a term that we wish to substitute for.

Recall from existential introduction:

- Matching the top of our rule, $P = Q(x_0, x)$, so line 3 of the proof is $P[x_0 / x]$, which is $Q(x_0, x)$

- So we don’t have to substitute in for every occurrence of a term.
Theory of Equality
- Examples

From these two inference rules, we can derive two other properties that we expect equality to have:

- Symmetry : $\vdash_{ND} \forall x, y. (x = y) \Rightarrow (y = x)$
- Transitivity : $\vdash_{ND} \forall x, y, z. (x = y) \land (y = z) \Rightarrow (x = z)$

Theory of Equality
- Example

$\vdash_{ND} \forall x, y. (x = y) \Rightarrow (y = x)$

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_0$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$y_0$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$x_0 = y_0$</td>
<td>assumption</td>
</tr>
<tr>
<td>4</td>
<td>$x_0 = x_0$</td>
<td>$i$</td>
</tr>
<tr>
<td>5</td>
<td>$y_0 = x_0$</td>
<td>$\epsilon 3, 4$</td>
</tr>
<tr>
<td>6</td>
<td>$x_0 = y_0 \Rightarrow y_0 = x_0$</td>
<td>$\Rightarrow i 3 - 5$</td>
</tr>
<tr>
<td>7</td>
<td>$\forall y. (x_0 = y) \Rightarrow (y = x_0)$</td>
<td>$\forall 1 2 - 6$</td>
</tr>
<tr>
<td>8</td>
<td>$\forall x, y. (x = y) \Rightarrow (y = x)$</td>
<td>$\forall 1 - 7$</td>
</tr>
</tbody>
</table>
Theory of Equality
- Example

\[ \forall x, y, z. (x = y) \land (y = z) \Rightarrow (x = z) \]

1. \[ x_0 \]
2. \[ y_0 \]
3. \[ (x_0 = y_0) \land (y_0 = z_0) \quad \text{assumption} \]
4. \[ x_0 = y_0 \] \quad \land \text{e 3} \]
5. \[ y_0 = z_0 \] \quad \land \text{e 3} \]
6. \[ x_0 = z_0 \] \quad = \text{e 4, 5} \]
7. \[ (x_0 = y_0) \land (y_0 = z_0) \Rightarrow (x_0 = z_0) \Rightarrow i 3 - 6 \]
8. \[ \forall y. (x_0 = y) \land (y = z_0) \Rightarrow (x_0 = z_0) \quad \forall i 2 - 7 \]
9. \[ \forall x, y. (x = y) \land (y = z) \Rightarrow (x = z) \quad \forall i 1 - 8 \]

Theory of Equality
- Leibniz’s Law

- The substitution inference rule is related to Leibniz’s Law.
- **Leibniz’s Law:**
  
  if \( t_1 = t_2 \) is a theorem, then so is \( P[t_1 / x] \Leftrightarrow P[t_2 / x] \)

- Leibniz’s Law is generally referred to as the ability to substitute “equals for equals”.
Leibniz

Gottfried Wilhelm von Leibniz (1646-1716)
- The founder of differential and integral calculus.
- Another of Leibniz’s lifelong aims was to collate all human knowledge.

"[He was] one of the last great polymaths - not in the frivolous sense of having a wide general knowledge, but in the deeper sense of one who is a citizen of the whole world of intellectual inquiry."

Theory of Equality
- Example

- From our natural deduction rules, we can derive Leibniz’s Law:

\[ t_1 = t_2 \vdash_{\text{ND}} P(t_1) \iff P(t_2) \]
Theory of Equality
- Equality: Semantics

- The semantics of the equality symbol is equality on the objects of the domain.
- In ALL interpretations it means the same thing.
- Normal interpretations are interpretations in which the symbol = is interpreted as designating the equality relation on the domain.
- We will restrict ourselves to normal interpretations from now on.

Theory of Equality
- Extensional Equality

- Equality in the domain is extensional, meaning it is equality in meaning rather than form.
- This is in contrast to intensional equality which is equality in form rather than meaning.
- In logic, we are interested in whether two terms represent the same object, not whether they are the same symbols.
- If two terms are intensionally equal then they are also extensionally equal, but not necessarily the other way around.
Theory of Equality
- Equality: Counterexamples

- Show the following argument is not valid:
  \[ \exists x. P(x) \land Q(x), P(A), A = B \models Q(B) \]
- where \(A, B\) are constants

Theory of Arithmetic

- Another commonly used theory is that of arithmetic.
- It was formalized by Dedekind in 1879 and also by Peano in 1889.
- It is generally referred to as Peano’s Axioms.
- The model of the system is the natural numbers with the constants 0 and 1, the functions +, \(\ast\), and the relation <.
Peano’s Axioms

P1: \( \forall x. \forall y. (x + y = y + x) \)  
Commutativity of +

P2: \( \forall x. \forall y. (x * y = y * x) \)  
Commutativity of *

P3: \( \forall x. \forall y. \forall z. x + (y + z) = (x + y) + z \)  
Associativity of +

P4: \( \forall x. \forall y. \forall z. x * (y * z) = (x * y) * z \)  
Associativity of *

P5: \( \forall x. \forall y. \forall z. x * (y + z) = (x * y) + (x * z) \)  
Distributivity

P6: \( x + 0 = x \)  
Property of 0

P7: \( x * 1 = x \)  
Property of 1

P8: \( \forall x. \neg (x + 1 = 0) \)  
0 is not a successor

P9: \( \forall x. \forall y. x + 1 = y + 1 \Rightarrow x = y \)

P10: \( \forall x. \forall y. x < y \Rightarrow \exists z. (z = 0) \land y = x + z \)  
Property of <

P11: \( P[0/x] \land (\forall y. P[y/x] \Rightarrow P[y + 1/x]) \Rightarrow \forall x. P \)  
Induction Scheme

Intuitionistic Logic

- “A proof that something exists is constructive if it provides a method for actually constructing it.”
- In intuitionistic logic, only constructive proofs are allowed.
- Therefore, they disallow proofs by contradiction. To show \( \phi \), you can’t just show \( \neg \phi \) is impossible.
- They also disallow the law of the excluded middle arguing that you have to actually show one of \( \phi \) or \( \neg \phi \) before you can conclude \( \phi \lor \neg \phi \)
- Intuitionistic logic was invented by Brouwer. Theorem provers that use intuitionistic logic are Nuprl, Coq, Elf, and Lego.
- In this course, we will only be studying classical logic.
Summary

- Predicate Logic (motivation, syntax and terminology, semantics, axiom systems, natural deduction)
- Equality, Arithmetic
- Mechanical theorem proving

Theorem proving
Formal Methods
Lecture 8

Farn Wang
Department of Electrical Engineering
National Taiwan University
Theorem Proving: Historical Perspective

- Theorem proving (or automated deduction) = logical deduction performed by machine
- At the intersection of several areas
  - Mathematics: original motivation and techniques
  - Logic: the framework and the meta-reasoning techniques

Theorem proving

- Prove that an implementation satisfies a specification by mathematical reasoning
Theorem proving

- Implementation and specification expressed as formulas in a formal logic
- Required relationship (logical equivalence/logical implication) described as a theorem to be proven within the context of a proof calculus
- A proof system:
  - A set of axioms and inference rules (simplification, rewriting, induction, etc.)

Proof checking

- It is a purely syntactic matter to decide whether each theorem is an axiom or follows from previous theorems (axioms) by a rule of inference
Proof generation

- Complete automation generally impossible: theoretical undecidability limitations
- However, a great deal can be automated (decidable subsets, specific classes of applications and specification styles)

Applications

- Hardware and software verification (or debugging)
- Automatic program synthesis from specifications
- Discovery of proofs of conjectures
  - A conjecture of Tarski was proved by machine (1996)
  - There are effective geometry theorem provers
Program Verification

- Fact: mechanical verification of software would improve software productivity, reliability, efficiency
- Fact: such systems are still in experimental stage
  - After 40 years!
  - Research has revealed formidable obstacles
  - Many believe that program verification is extremely difficult

Program Verification

- Fact:
  - Verification is done with respect to a specification
  - Is the specification simpler than the program?
  - What if the specification is not right?
- Answer:
  - Developing specifications is hard
  - Still redundancy exposes many bugs as inconsistencies
  - We are interested in partial specifications
    - An index is within bounds, a lock is released...
Programs, Theorems. Axiomatic Semantics

- Consists of:
  - A language for writing specifications about programs
  - Rules for establishing when specifications hold

- Typical specifications:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with $x = 0$

- Partial vs. total correctness specifications
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)

Specification Languages

- Must be easy to use and expressive (conflicting needs)
  - Most often only expressive

- Typically they are extensions of first-order logic
  - Although higher-order or modal logics are also used

- We focus here on state-based specifications (safety)
  - State = values of variables + contents of heap (+ past state)

  - Not allowed: “variable x is live”, “lock L will be released”, “there is no correlation between the values of x and y”
A Specification Language

- We’ll use a fragment of first-order logic:
  - Formulas \( P ::= A \mid \text{true} \mid \text{false} \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid \neg P \mid \forall x.P \)
  - Atoms \( A ::= E_1 \leq E_2 \mid E_1 = E_2 \mid f(A_1,\ldots,A_n) \mid \ldots \)

- All boolean expressions from our language are atoms

- Can have an arbitrary collection of predicate symbols
  - \( \text{reachable}(E_1,E_2) \) - list cell \( E_2 \) is reachable from \( E_1 \)
  - \( \text{sorted}(a, L, H) \) - array \( a \) is sorted between \( L \) and \( H \)
  - \( \text{ptr}(E,T) \) - expression \( E \) denotes a pointer to \( T \)
  - \( E : \text{ptr}(T) \) - same in a different notation

Program Verification Using Hoare’s Logic
Hoare Triples

- Partial correctness: \( \{ P \} s \{ Q \} \)
  - When you start \( s \) in any state that satisfies \( P \)
  - If the execution of \( s \) terminates
  - It does so in a state that satisfies \( Q \)
- Total correctness: \([ P ] s [ Q ]\)
  - When you start \( s \) in any state that satisfies \( P \)
  - The execution of \( s \) terminates and
  - It does so in a state that satisfies \( Q \)
- Defined inductively on the structure of statements

Hoare Rules

- Assignments
  - \( y := t \)
- Composition
  - \( S1; S2 \)
- If-then-else
  - \( \text{if } e \text{ the } S1 \text{ else } S2 \)
- While
  - \( \text{while } e \text{ do } S \)
- Consequence
Greatest common divisor

\[
\{x_1 > 0 \land x_2 > 0\}
\]
\[
y_1 := x_1;
\]
\[
y_2 := x_2;
\]
while \(\neg (y_1 = y_2)\) do
  if \(y_1 > y_2\) then \(y_1 := y_1 - y_2\)
  else \(y_2 := y_2 - y_1\)
\[
\{y_1 = \gcd(x_1, x_2)\}
\]

Why it works?

- Suppose that \(y_1, y_2\) are both positive integers.
  - If \(y_1 > y_2\) then \(\gcd(y_1, y_2) = \gcd(y_1 - y_2, y_2)\)
  - If \(y_2 > y_1\) then \(\gcd(y_1, y_2) = \gcd(y_1, y_2 - y_1)\)
  - If \(y_1 - y_2\) then \(\gcd(y_1, y_2) = y_1 = y_2\)
Hoare Rules: Assignment

- General rule:
  - \{p[t/y]\} y:=t \{p\}

- Examples:
  - \{y+5=10\} y:=y+5 \{y=10\}
  - \{y+y<z\} x:=y \{x+y<z\}
  - \{2*(y+5)>20\} y:=2*(y+5) \{y>20\}

- Justification: write \(p\) with \(y'\) instead of \(y\), and add the conjunct \(y'=t\). Next, eliminate \(y'\) by replacing \(y'\) by \(t\).

---

{p} y:=t \{?\}

- Strategy: write \(p\) and the conjunct \(y=t\), where \(y'\) replaces \(y\) in both \(p\) and \(t\). Eliminate \(y'\).

Example:
\{y>5\} y:=2*(y+5) \{?\}

- {p} y:=t \{\exists y' (p[y'/y] \land t[y'/y]=y)\}

\(y'>5 \land y=2*(y'+5) \rightarrow y>20\)

---
**Hoare Rules: Composition**

- **General rule:**
  - \( \{p\} S1 \{r\}, \{r\} S2 \{q\} \rightarrow \{p\} S1;S2 \{q\} \)

- **Example:**
  
  if the antecedents are
  1. \( \{x+1=y+2\} \ x:=x+1 \ \{x=y+2\} \)
  2. \( \{x=y+2\} \ y:=y+2 \ \{x=y\} \)
  
  Then the consequent is
  \( \{x+1=y+2\} \ x:=x+1; \ y:=y+2 \ \{x=y\} \)

---

**Hoare Rules: If-then-else**

- **General rule:**
  - \( \{p \wedge e\} S1 \{q\}, \{p \wedge \neg e\} S2 \{q\} \)
  - \( \{p\} \text{ if } e \text{ then } S1 \text{ else } S2 \{q\} \)

- **Example:**
  
  \( p \) is \( \gcd(y_1,y_2)=\gcd(x_1,x_2) \wedge y_1>0 \wedge y_2>0 \wedge \neg(y_1=y_2) \)
  
  \( e \) is \( y_1>y_2 \)
  
  \( S1 \) is \( y_1:=y_1-y_2 \)
  
  \( S2 \) is \( y_2:=y_2-y_1 \)
  
  \( q \) is \( \gcd(y_1,y_2)=\gcd(x_1,x_2) \wedge y_1>0 \wedge y_2>0 \)
Hoare Rules: While

- General rule:
  - \{p \land e\} S \{p\}
  - \{p\} while e do S \{p \land \neg e\}

- Example:
  - \(p\) is \{gcd(y_1, y_2) = gcd(x_1, x_2) \land y_1 > 0 \land y_2 > 0\}
  - \(e\) is \((y_1 \neq y_2)\)
  - \(S\) is if \(y_1 > y_2\) then \(y_1 := y_1 - y_2\) else \(y_2 := y_2 - y_1\)

Hoare Rules: Consequence

- Strengthen a precondition
  - \(r \rightarrow p, \{p\} S \{q\}\)
  - \(\{r\} S \{q\}\)

- Weaken a postcondition
  - \(\{p\} S \{q\}, q \rightarrow r\)
  - \(\{p\} S \{r\}\)
Soundness

- Hoare logic is sound in the sense that everything that can be proved is correct!
- This follows from the fact that each axiom and proof rule preserves soundness.

Completeness

- A proof system is called complete if every correct assertion can be proved.
- Propositional logic is complete.
- No deductive system for the standard arithmetic can be complete (Godel).
And for Hoare logic?

- Let $S$ be a program and $p$ its precondition.
- Then $\{p\} S \{false\}$ means that $S$ never terminates when started from $p$. This is undecidable. Thus, Hoare’s logic cannot be complete.

Hoare Rules: Examples

- Consider
  - $\{x = 2\} x := x + 1 \{x < 5\}$
  - $\{x < 2\} x := x + 1 \{x < 5\}$
  - $\{x < 4\} x := x + 1 \{x < 5\}$

- They all have correct preconditions
- But the last one is the most general (or weakest) precondition
Dijkstra’s Weakest Preconditions

- Consider \( \{ P \} s \{ Q \} \)
- Predicates form a lattice:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>valid preconditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strong</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- To verify \( \{ P \} s \{ Q \} \)
- compute \( WP(s, Q) \) and prove \( P \neq WP(s, Q) \)

Weakest predition, Strongest postcondition

- For an assertion \( p \) and code \( S \), let \( \text{post}(p,S) \) be the strongest assertion such that \( \{p\}S\{\text{post}(p,S)\} \)
  That is, if \( \{p\}S\{q\} \) then \( \text{post}(p,S) \rightarrow q \).

- For an assertion \( q \) and code \( S \), let \( \text{pre}(S,q) \) be the weakest assertion such that \( \{\text{pre}(S,q)\}S\{q\} \)
  That is, if \( \{p\}S\{q\} \) then \( p \rightarrow \text{pre}(S,q) \).
Relative completeness

- Suppose that either
  - post(p,S) exists for each p, S, or
  - pre(S,q) exists for each S, q.
- Some oracle decides on pure implications.
  Then each correct Hoare triple can be proved.

What does that mean? The weakness of the proof system stem from the weakness of the (FO) logic, not of Hoare’s proof system.

Extensions

Many extensions for Hoare’s proof rules:
- Total correctness
- Arrays
- Subroutines
- Concurrent programs
- Fairness
Higher-Order Logic

- **First-order logic:**
  - only domain variables can be quantified.

- **Second-order logic:**
  - quantification over subsets of variables (i.e., over predicates).

- **Higher-order logics:**
  - quantification over arbitrary predicates and functions.
Higher-Order Logic

- Variables can be functions and predicates,
- Functions and predicates can take functions as arguments and return functions as *values*,
- Quantification over functions and predicates.

Since arguments and results of predicates and functions can themselves be predicates or functions, this imparts a **first-class status** to functions, and allows them to be manipulated just like *ordinary values*

### Higher-Order Logic

**Example 1:** (mathematical induction)

\[
\forall P. \ [P(0) \land (\forall n. P(n) \rightarrow P(n+1))] \rightarrow \forall n. P(n)
\]

(Impossible to express it in FOL)

**Example 2:**

Function Rise defined as \(\text{Rise}(c, t) = \neg c(t) \land c(t+1)\)

Rise expresses the notion that a signal \(c\) rises at time \(t\).

Signal is modeled by a function \(c: \mathbb{N} \rightarrow \{F, T\}\), passed as argument to Rise.

Result of applying Rise to \(c\) is a function: \(\mathbb{N} \rightarrow \{F, T\}\).
Higher-Order Logic (cont’d)

- **Advantage:**
  - high expressive power!

- **Disadvantages:**
  - Incompleteness of a sound proof system for most higher-order logics
  - **Theorem** (Gödel, 1931)
    
    There is no complete deduction system for the second-order logic.
    
  - Reasoning more difficult than in FOL, need ingenious inference rules and heuristics.

---

Higher-Order Logic (cont’d)

- **Disadvantages:**
  - Inconsistencies can arise in higher-order systems if semantics not carefully defined
    
    **“Russell Paradox”:**
    
    Let P be defined by \( P(Q) = \neg Q(Q) \). By substituting P for Q, leads to \( P(P) = \neg P(P) \).
    
    **Contradiction!**
    
    (P: bool \( \rightarrow \) bool, Q: bool \( \rightarrow \) bool)
  
  - Introduction of “types” (syntactical mechanism) is effective against certain inconsistencies.
  - Use controlled form of logic and inferences to minimize the risk of inconsistencies, while gaining the benefits of powerful representation mechanism.
  - Higher-order logic increasingly popular for hardware verification!
Theorem Proving Systems

- Automated deduction systems (e.g. Prolog)
  - full automatic, but only for a decidable subset of FOL
  - speed emphasized over versatility
  - often implemented by ad hoc decision procedures
  - often developed in the context of AI research

- Interactive theorem proving systems
  - semi-automatic, but not restricted to a decidable subset
  - versatility emphasized over speed
  - in principle, a complete proof can be generated for every theorem

Some theorem proving systems:
- Boyer-Moore (first-order logic)
- HOL (higher-order logic)
- PVS (higher-order logic)
- Lambda (higher-order logic)
HOL

- HOL (Higher-Order Logic) developed at University of Cambridge
- Interactive environment (in ML, Meta Language) for machine assisted theorem proving in higher-order logic (a proof assistant)
- Steps of a proof are implemented by applying inference rules chosen by the user; HOL checks that the steps are safe
- All inferences rules are built on top of eight primitive inference rules

HOL

- Mechanism to carry out backward proofs by applying built-in ML functions called tactics and tacticals
- By building complex tactics, the user can customize proof strategies
- Numerous applications in software and hardware verification
- Large user community
HOL Theorem Prover

- Logic is strongly typed (type inference, abstract data types, polymorphic types, etc.)
- It is sufficient for expressing most ordinary mathematical theories (the power of this logic is similar to set theory)
- HOL provides considerable built-in theorem-proving infrastructure:
  - a powerful rewriting subsystems
  - library facility containing useful theories and tools for general use
  - Decision procedures for tautologies and semi-decision procedure for linear arithmetic provided as libraries

HOL Theorem Prover

- The primary interface to HOL is the functional programming language ML
- Theorem proving tools are functions in ML (users of HOL build their own application specific theorem proving infrastructure by writing programs in ML)
- Many versions of HOL:
  - HOL88: Classic ML (from LCF);
  - HOL90: Standard ML
  - HOL98: Moscow ML
HOL Theorem Prover (cont’d)

- HOL and ML

HOL = Some predefined functions + types

The ML Language

- The HOL systems can be used in two main ways:
  - for directly proving theorems: when higher-order logic is a suitable specification language (e.g., for hardware verification and classical mathematics)
  - as embedded theorem proving support for application-specific verification systems when specification in specific formalisms needed to be supported using customized tools.

- The approach to mechanizing formal proof used in HOL is due to Robin Milner. He designed a system, called LCF: Logic for Computable Functions. (The HOL system is a direct descendant of LCF.)

Specification in HOL

- **Functional description:**
  express output signal as function of input signals, e.g.:
  
  \[ \text{AND gate:} \]
  \[
  \text{out} = \text{and} (\text{in}1, \text{in}2) = (\text{in}1 \land \text{in}2)
  \]

- **Relational (predicate) description:**
  gives relationship between inputs and outputs in the form of a predicate (a Boolean function returning “true” of “false”), e.g.:
  
  \[ \text{AND gate:} \]
  \[
  \text{AND} ((\text{in}1, \text{in}2),\text{(out)}) : \text{out} = (\text{in}1 \land \text{in}2)
  \]
**Specification in HOL**

**Notes:**
- functional descriptions allow recursive functions to be described. They cannot describe bi-directional signal behavior or functions with multiple feedback signals, though
- relational descriptions make no difference between inputs and outputs
- Specification in HOL will be a combination of predicates, functions and abstract types

---

**Specification in HOL**

### Network of modules

- conjunction "\( \land \)" of implementation module predicates
  \[
  M(a, b, c, d, e) := M1(a, b, p, q) \land M2(q, b, e) \land M3(e, p, c, d)
  \]
- hide internal lines \((p,q)\) using **existential quantification**
  \[
  M(a, b, c, d, e) := \exists p q. M1(a, b, p, q) \land M2(q, b, e) \land M3(e, p, c, d)
  \]
Specification in HOL

Combinational circuits

\[
\text{SPEC (in1, in2, in3, in4, out):=}
\]
\[
\text{out = (in1 } \land \text{ in2) } \lor \text{ (in3 } \land \text{ in4)}
\]

\[
\text{IMPL (in1, in2, in3, in4, out):=}
\]
\[
\exists l1, l2. \text{ AND (in1, in2, l1) } \land \text{ AND (in3, in4, l2) } \lor \text{ OR (l1, l2, out)}
\]
\[
\text{where AND (a, b, c):= (c } = \text{ a } \land \text{ b)}
\]
\[
\text{OR (a, b, c):= (c } = \text{ a } \lor \text{ b)}
\]

Specification in HOL

- **Note**: a functional description would be:

\[
\text{IMPL (in1, in2, in3, in4, out):=}
\]
\[
\text{out = (or (and (in1, in2), and (in3, in4))}
\]
\[
\text{where and (in1, in2) = (in1 } \land \text{ in2)}
\]
\[
\text{or (in1, in2) = (in1 } \lor \text{ in2)}
\]
Specification in HOL

Sequential circuits
- Explicit expression of time (discrete time modeled as natural numbers).
- Signals defined as functions over time, e.g. type: \( \text{nat} \rightarrow \text{bool} \) or \( \text{nat} \rightarrow \text{bitvec} \)
- Example: D-flip-flop (latch):
  \[
  \text{DFF (in, out)} := (\text{out (0) = F}) \land (\forall t. \text{out (t+1) = in (t)})
  \]
  \( \text{in} \) and \( \text{out} \) are functions of time \( t \) to boolean values: type \( \text{nat} \rightarrow \text{bool} \)

Specification in HOL

- Notion of time can be added to combinational circuits, e.g., AND gate
  \[
  \text{AND (in1, in2, out)} := \forall t. \text{out (t) = (in1(t) \land in2(t))}
  \]

- Temporal operators can be defined as predicates, e.g.:
  \[
  \text{EVENTUAL sig } t1 = \exists t2. (t2 > t1) \land \text{sig t2}
  \]
  meaning that signal “sig” will eventually be true at time \( t2 > t1 \).

- Note: This kind of specification using existential quantified time variables is useful to describe asynchronous behavior
A formal proof is a sequence, each of whose elements is
- either an *axiom*
- or follows from earlier members of the sequence by a *rule of inference*

A *theorem* is the last element of a proof

A *sequent* is written:
- $\Gamma \vdash P$, where $\Gamma$ is a *set of assumptions* and $P$ is the *conclusion*

In HOL, this consists in applying ML functions representing rules of inference to axioms or previously generated theorems

The sequence of such applications directly correspond to a proof

A value of *type thm* can be obtained either
- directly (as an axiom)
- by computation (using the built-in functions that represent the inference rules)

ML typechecking ensures these are the only ways to generate a thm:
*All theorems must be proved!*
Verification Methodology in HOL

1. Establish a formal specification (predicate) of the intended behavior (SPEC)
2. Establish a formal description (predicate) of the implementation (IMP), including:
   - behavioral specification of all sub-modules
   - structural description of the network of sub-modules
3. Formulation of a proof goal, either
   - IMP $\Rightarrow$ SPEC (proof of implication), or
   - IMP $\Leftrightarrow$ SPEC (proof of equivalence)
4. Formal verification of above goal using a set of inference rules

Example 1: Logic AND

- **AND Specification:**
  - AND_SPEC (i1,i2,out) := out = i1 $\land$ i2

- **NAND specification:**
  - NAND (i1,i2,out) := out = $\neg$(i1 $\land$ i2)

- **NOT specification:**
  - NOT (i, out) := out = $\neg$ i

- **AND Implementation:**
  - AND_IMPL (i1,i2,out) := $\exists x$. NAND (i1,i2,x) $\land$ NOT (x,out)
**Example 1: Logic AND**

- **Proof Goal:**
  - \( \forall i1, i2, out. \text{AND_IMPL}(i1,i2,out) \Rightarrow \text{ANDSPEC}(i1,i2,out) \)

- **Proof (forward)**
  \( \text{AND_IMPL}(i1,i2,out) \) (from above circuit diagram)
  - \( \exists x. \text{NAND}(i1,i2,x) \land \text{NOT}(x,out) \) (by def. of AND impl)
  - \( \text{NAND}(i1,i2,x) \land \text{NOT}(x,out) \) (strip off "\( \exists x. \)"")
  - \( \text{NAND}(i1,i2,x) \) (left conjunct of line 3)
  - \( x = \neg (i1 \land i2) \) (by def. of NAND)
  - \( \text{NOT}(x,out) \) (right conjunct of line 3)
  - \( \text{out} = \neg x \) (by def. of NOT)
  - \( \text{out} = \neg (\neg (i1 \land i2) \) (substitution, line 5 into 7)
  - \( \text{out} = (i1 \land i2) \) (simplify, \( \neg \neg t = t \))
  - \( \text{AND}(i1,i2,out) \) (by def. of AND spec)
  - \( \text{AND_IMPL}(i1,i2,out) \Rightarrow \text{AND_SPEC}(i1,i2,out) \)
  Q.E.D.

---

**Example 2: CMOS-Inverter**

- **Specification** (black-box behavior)
  - \( \text{Spec}(x,y) := (y = \neg x) \)

- **Implementation**

  ![CMOS-Inverter Circuit Diagram]

  - **Basic Modules Specs**
    - \( \text{PWR}(x) := (x = T) \)
    - \( \text{GND}(x) := (x = F) \)
    - \( \text{N-Trans}(g,x,y) := (g \Rightarrow (x = y)) \)
    - \( \text{P-Trans}(g,x,y) := (\neg g \Rightarrow (x = y)) \)
Example 2: CMOS-Inverter

- **Implementation (network structure)**
  - Impl(x,y):= \exists p, q.
    - PWR(p) \land
    - GND(q) \land
    - N-\text{Tran}(x,y,q) \land
    - P-\text{Tran}(x,p,y)

- **Proof goal**
  - \forall x, y. \text{Impl}(x,y) \iff \text{Spec}(x,y)

- **Proof (forward)**
  - Impl(x,y):= \exists p, q.
    - (p = T) \land
    - (q = F) \land
    - (\text{substitution of the definition of PWR and GND})
    - N-\text{Tran}(x,y,q) \land
    - P-\text{Tran}(x,p,y)
Example 2: CMOS-Inverter

- \( \text{Impl}(x,y) := \)
  - \( T \land T \)  
  - \( \text{N-Tran}(x,y,F) \land \text{P-Tran}(x,T,y) \) (use Thm: "(\( \exists a. a=T \) \( = T \) and (\( \exists a. a=F \) \( = T \))")

- \( \text{Impl}(x,y) := \)
  - \( \text{N-Tran}(x,y,F) \land \text{P-Tran}(x,T,y) \) (use Thm: "\( x \land T = x \)"

- \( \text{Impl}(x,y) := \)
  - \( (x \Rightarrow (y = F)) \land (\neg x \Rightarrow (T = y)) \) (use def. of N-Tran and P-Tran)

Boolean simplifications:
- \( \text{Impl}(x,y) := (\neg x \lor (y = F)) \land ((a \Rightarrow b) = (\neg a \lor b)) \)
- \( \text{Impl}(x,y) := F \lor (\neg x \land (T = y)) \lor ((y = F) \land x) \lor F \)
- \( \text{Impl}(x,y) := (\neg x \land (T = y)) \lor ((y = F) \land x) \)
Example 2: CMOS-Inverter

- Case analysis $x=T/F$
  
  - $x=T$: $\text{Impl}(T,y) := (F \land (T = y)) \lor ((y = F) \land T)$
  
  - $x=F$: $\text{Impl}(F,y) := (T \land (T = y)) \lor ((y = F) \land F)$

  $\begin{align*}
  x=T: & \text{Impl}(T,y) := (y = F) \\
  x=F: & \text{Impl}(F,y) := (T = y)
  \end{align*}$

- Case analysis on Spec:
  
  - $x=T$: $\text{Spec}(T,y) := (y = F)$
  
  - $x=F$: $\text{Spec}(F,y) := (y = T)$

- Conclusion: $\neg \text{Spec}(x,y) \iff \text{Impl}(x,y)$

Abstraction Forms

- **Structural abstraction:**
  
  - only the behavior of the external inputs and outputs of a module is of interest (abstracts away any internal details)

- **Behavioral abstraction:**
  
  - only a specific part of the total behavior (or behavior under specific environment) is of interest

- **Data abstraction:**
  
  - behavior described using abstract data types (e.g. natural numbers instead of Boolean vectors)

- **Temporal abstraction:**
  
  - behavior described using different time granularities (e.g. refinement of instruction cycles to clock cycles)
Example 3: 1-bit Adder

- **Specification:**
  - ADDER_SPEC (in1: nat, in2: nat, cin: nat, sum: nat, cout: nat) :=
    in1 + in2 + cin = 2 * cout + sum

- **Implementation:**

- **Note:** Spec is a structural abstraction of Impl.

1-bit Adder (cont’d)

- **Implementation:**
  
  ADDER_IMPL(in1: bool, in2: bool, cin: bool, sum: bool, cout: bool) :=
  ∃ l1 l2 l3.
  EXOR (in1, in2, l1) ∧
  AND (in1, in2, l2) ∧
  EXOR (l1, cin, sum) ∧
  AND (l1, cin, l3) ∧
  OR (l2, l3, cout)

- Define a data abstraction function (bn: bool → nat) needed to relate Spec variable types (nat) to Impl variable types (bool):

  \[
  bn(x) := \begin{cases} 
  1, & \text{if } x = T \\
  0, & \text{if } x = F 
  \end{cases}
  \]
1-bit Adder (cont’d)

- **Proof goal:**
  \[ \forall \text{in1}, \text{in2}, \text{cin}, \text{sum, cout}. \]
  \[ \text{ADDER_impl (in1, in2, cin, sum, cout)} \]
  \[ \Rightarrow \text{ADDER_spec} (b\text{n}(\text{in1}), b\text{n}(\text{in2}), b\text{n}(\text{cin}), b\text{n}(\text{sum}), b\text{n}(\text{cout})) \]

Verification of Generic Circuits

- used in datapath design and verification
- **idea:**
  - verify \( n \)-bit circuit then specialize proof for specific value of \( n \), (i.e., once proven for \( n \), a simple instantiation of the theorem for any concrete value, e.g. 32, gets a proven theorem for that instance).
  - use of induction proof
Example 4: N-bit Adder

- **N-bit Adder**

- **Specification**
  - N-ADDER_SPEC (n,in1,in2,cin,sum,cout):=
    - \((in1 + in2 + cin = 2^{n+1} * cout + sum)\)

---

Example 4: N-bit Adder

- **Implementation**
Implementation

- recursive definition:
  \[
  \text{N-ADDER_IMP}(n, \text{in1}[0..n-1], \text{in2}[0..n-1], \text{cin}, \text{sum}[0..n-1], \text{cout}) := \\
  \exists w. \text{N-ADDER_IMP}(n-1, \text{in1}[0..n-2], \text{in2}[0..n-2], \text{cin}, \text{sum}[0..n-2], w) \land \\
  \text{N-ADDER_IMP}(1, \text{in1}[n-1], \text{in2}[n-1], w, \text{sum}[n-1], \text{cout})
  \]

- Note:
  - \( \text{N-ADDER_IMP}(1, \text{in1}[i], \text{in2}[i], \text{cin}, \text{sum}[i], \text{cout}) = \text{ADDER_IMP}(\text{in1}[i], \text{in2}[i], \text{cin}, \text{sum}[i], \text{cout}) \)

- Data abstraction function \((\text{vn} : \text{bitvec} \rightarrow \text{nat})\) to relate bit vectors to natural numbers:
  - \( \text{vn}(x[0]) := b_0(x[0]) \)
  - \( \text{vn}(x[0,n]) := 2^n \cdot b_n(x[n]) + \text{vn}(x[0,n-1]) \)

---

**Proof goal:**

\[
\forall n, \text{in1}, \text{in2}, \text{cin}, \text{sum}, \text{cout}.
\text{N-ADDER_IMP}(n, \text{in1}[0..n-1], \text{in2}[0..n-1], \text{cin}, \text{sum}[0..n-1], \text{cout}) \\
\Rightarrow \text{N-ADDER_SPEC}(n, \text{vn}(\text{in1}[0..n-1]), \text{vn}(\text{in2}[0..n-1]), \text{vn}(\text{cin}), \\
\text{vn}(\text{sum}[0..n-1]), \text{vn}(\text{cout}))
\]

- can be **instantiated with** \( n = 32 \):

\[
\forall \text{in1}, \text{in2}, \text{cin}, \text{sum}, \text{cout}.
\text{N-ADDER_IMP}(\text{in1}[0..31], \text{in2}[0..31], \text{cin}, \text{sum}[0..31], \text{cout}) \\
\Rightarrow \text{N-ADDER_SPEC}(\text{vn}(\text{in1}[0..31]), \text{vn}(\text{in2}[0..31]), \text{vn}(\text{cin}), \\
\text{vn}(\text{sum}[0..31]), \text{vn}(\text{cout}))
\]
Proof by induction over n:

- basis step:
  \[ \text{N-ADDER_IMP}(0, \text{in1}[0], \text{in2}[0], \text{cin}, \text{sum}[0], \text{cout}) \]
  \[ \Rightarrow \text{N-ADDER_SPEC}(0, \text{vn}(\text{in1}[0]), \text{vn}(\text{in2}[0]), \text{vn}(\text{cin}), \text{vn}(\text{sum}[0]), \text{vn}(\text{cout})) \]

- induction step:
  \[ \left[ \text{N-ADDER_IMP}(n, \text{in1}[0..n-1], \text{in2}[0..n-1], \text{cin}, \text{sum}[0..n-1], \text{cout}) \right] \]
  \[ \Rightarrow \left[ \text{N-ADDER_SPEC}(n, \text{vn}(\text{in1}[0..n-1]), \text{vn}(\text{in2}[0..n-1]), \text{vn}(\text{cin}), \text{vn}(\text{sum}[0..n-1]), \text{vn}(\text{cout})) \right] \]

Notes:

- basis step is equivalent to 1-bit adder proof, i.e.
  \[ \text{ADDER_IMP}(\text{in1}[0], \text{in2}[0], \text{cin}, \text{sum}[0], \text{cout}) \]
  \[ \Rightarrow \text{ADDER_SPEC}(\text{bn}(\text{in1}[0]), \text{bn}(\text{in2}[0]), \text{bn}(\text{cin}), \text{bn}(\text{sum}[0]), \text{bn}(\text{cout})) \]

- induction step needs more creativity and work load!
Practical Issues of Theorem Proving

No fully automatic theorem provers. All require human guidance in indirect form, such as:

- When to delete redundant hypotheses, when to keep a copy of a hypothesis
- Why and how (order) to use lemmas, what lemma to use is an art
- How and when to apply rules and rewrites
- Induction hints (also nested induction)

Practical Issues of Theorem Proving

- Selection of proof strategy, orientation of equations, etc.
- Manipulation of quantifiers (forall, exists)
- Instantiation of specification to a certain time and instantiating time to an expression
- Proving lemmas about (modulus) arithmetic
- Trying to prove a false lemma may be long before abandoning
Prototype Verification System (PVS)

- Provides an integrated environment for the development and analysis of formal specifications.
- Supports a wide range of activities involved in creating, analyzing, modifying, managing, and documenting theories and proofs.
Prototype Verification System (cont’)

- The primary purpose of PVS is to provide formal support for conceptualization and debugging in the early stages of the lifecycle of the design of a hardware or software system.

- In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable.

- The primary emphasis in the PVS proof checker is on supporting the construction of readable proofs.

- In order to make proofs easier to develop, the PVS proof checker provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas.
The PVS Language

- The specification language of PVS is built on higher-order logic
  - Functions can take functions as arguments and return them as values
  - Quantification can be applied to function variables
- There is a rich set of built-in types and type-constructors, as well as a powerful notion of subtype.
- Specifications can be constructed using definitions or axioms, or a mixture of the two.

The PVS Language (cont’)

- Specifications are logically organized into parameterized *theories* and *datatypes*.
- Theories are linked by *import* and *export* lists.
- Specifications for many foundational and standard theories are preloaded into PVS as prelude theories that are always available and do not need to be explicitly imported.
A Brief Tour of PVS

- Creating the specification
- Parsing
- Typechecking
- Proving
- Status
- Generating LATEX

A Simple Specification Example

```
sum: Theory
BEGIN
  n: VAR nat
  sum(n): RECURSIVE nat =
  (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF)
  MEASURE (LAMBDA n : n)
  closed_form: THEOREM sum(n) = (n * (n + 1)) / 2
END sum
```
Creating the Specification

- Create a file with a `.pvs` extension
  - Using the M-x new-pvs-file command (M-x nf) to create a new PVS file, and typing sum when prompted. Then type in the sum specification.
  - Since the file is included on the distribution tape in the Examples/tutorial subdirectory of the main PVS directory, it can be imported with the M-x import-pvs-file command (M-x imf). Use the M-x whereis-pvs command to find the path of the main PVS directory.
  - Finally, any external means of introducing a file with extension `.pvs` into the current directory will make it available to the system. ex: using vi.

Parsing

- Once the sum specification is displayed, it can be parsed with the `M-x parse (M-x pa)` command, which creates the internal abstract representation for the theory described by the specification.
- If the system finds an error during parsing, an error window will pop up with an error message, and the cursor will be placed in the vicinity of the error.
Typechecking

- To typecheck the file by typing M-x typecheck (M-x tc, C-c t), which checks for semantic errors, such as undeclared names and ambiguous types.
- Typechecking may build new files or internal structures such as TCCs. (when sum has been typechecked, a message is displayed in the minibuffer indicating the two TCCs were generated)

Typechecking (cont’)

- These TCCs represent proof obligations that must be discharged before the sum theory can be considered typechecked.
- TCCs can be viewed using the M-x show-tccs command.
Typechecking (cont’)

% Subtype TCC generated (line 7) for n-1
% unchecked
sum_TCC1: OBLIGATION (FORALL (n : nat) : NOT n=0 IMPLIES n-1 >= 0);

% Termination TCC generated (line 7) for sum
% unchecked
sum_TCC2: OBLIGATION (FORALL (n : nat) : NOT n=0 IMPLIES n-1 < n);

Typechecking (cont’)

- The first TCC is due to the fact that sum takes an argument of type nat, but the type of the argument in the recursive call to sum is integer, since nat is not closed under substraction.
  - Note that the TCC includes the condition NOT n=0, which holds in the branch of the IF-THEN-ELSE in which the expression n-1 occurs.
- The second TCC is needed to ensure that the function sum is total. PVS does not directly support partial functions, although its powerful subtyping mechanism allows PVS to express many operations that are traditionally regarded as partial.
  - The measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call.
Proving

- We are now ready to try to prove the main theorem
- Place the cursor on the line containing the closed form theorem and type M-x prove M-x pr or C-c p
- A new buffer will pop up the formula will be displayed and the cursor will appear at the Rule prompt indicating that the user can interact with the prover

Proving (cont’)

- First, notice the display, which consists of a single formula labeled {1} under a dashed line.
- This is a sequent: formulas above the dashed lines are called antecedents and those below are called succedents
  - The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the succedents
  - Either or both of the antecedents and succedents may be empty
The basic objective of the proof is to generate a proof tree in which all of the leaves are trivially true.

The nodes of the proof tree are sequents and while in the prover you will always be looking at an unproved leaf of the tree.

The current branch of a proof is the branch leading back to the root from the current sequent.

When a given branch is complete (i.e., ends in a true leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, notifies you that the proof is complete.

---

We will prove this formula by induction on $n$.

- To do this, type (induct "n")
- This generates two subgoals the one displayed is the base case where $n$ is 0
- To see the inductive step type (postpone) which postpones the current subgoal and moves on to the next unproved one Type (postpone) a second time to cycle back to the original subgoal (labeled closed_form.1)
To prove the base case, we need to expand the definition of sum, which is done by typing (expand "sum")

After expanding the definition of sum, we send the proof to the PVS decision procedures, which automatically decide certain fragments of arithmetic, by typing (assert)

This completes the proof of this subgoal and the system moves on to the next subgoal which is the inductive step

The first thing to do here is to eliminate the FORALL quantifier

This can most easily be done with the skolem! command, which provides new constants for the bound variables

To invoke this command type (skolem!) at the prompt

The resulting formula may be simplified by typing (flatten), which will break up the succedent into a new antecedent and succedent
Proving (cont’)

- The obvious thing to do now is to expand the definition of sum in the succedent. This again is done with the expand command, but this time we want to control where it is expanded, as expanding it in the antecedent will not help.
- So we type (expand “sum” +), indicating that we want to expand sum in the succedent.

Proving (cont’)

- The final step is to send the proof to the PVS decision procedures by typing (assert).
- The proof is now complete the system may ask whether to save the new proof and whether to display a brief printout of the proof.
closed_form :

| - · · · · · |
{1} (FORALL (n : nat) : sum(n) = (n * (n + 1)) / 2)

Rule? (induct "n")
Inducting on n,
this yields 2 subgoals

closed_form.1 :

| - · · · · · |
{1} sum(0) = (0 * (0 + 1)) / 2

Rule? (postpone)
Postponing closed_form.1

closed_form.2 :

| - · · · · · |
{1} (FORALL (j : nat) :
    sum(j) = (j * (j + 1)) / 2
    IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)

Rule? (postpone)
Postponing closed_form.2

closed_form.1 :

| - · · · · · |
{1} sum(0) = (0 * (0 + 1)) / 2

Rule? (expand "sum")
(IF 0 = 0 THEN 0 ELSE 0 + sum(0 - 1) ENDIF)
Expanding the definition of sum, this simplifies to:

closed_form.1:

```
|   -   -   -   -   -   -   -   |
{1}  0 = 0 / 2
```

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.1.

closed_form.2:

```
|   -   -   -   -   -   -   -   |
{1}  \forall (j : \text{nat}) :
    \text{sum}(j) = (j \cdot (j + 1)) / 2
    \Rightarrow \text{sum}(j + 1) = ((j + 1) \cdot (j + 1 + 1)) / 2
```

Rule? (skolem!)
Skolemizing,
this simplifies to:

closed_form.2:

```
|   -   -   -   -   -   -   -   |
{1}  \text{sum}(j ! 1) = (j ! 1 \cdot (j ! 1 + 1)) / 2
    \Rightarrow \text{sum}(j ! 1 + 1) = ((j ! 1 + 1) \cdot (j ! 1 + 1 + 1)) / 2
```

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
This simplifies to:

closed_form.2:

```
|   -   -   -   -   -   -   -   |
{1}  \text{sum}(j ! 1) = (j ! 1 \cdot (j ! 1 + 1)) / 2
|   -   -   -   -   -   -   -   |
{1}  \text{sum}(j ! 1 + 1) = ((j ! 1 + 1) \cdot (j ! 1 + 1 + 1)) / 2
```
Rule? (expand "sum" +)
(IF j ! 1 + 1 = 0 THEN 0 ELSE j ! 1 + 1 + sum(j ! 1 + 1 - 1) ENDIF)
simplifies to 1 + sum(j ! 1) + j ! 1
Expanding the definition of sum,
this simplifies to:
closed_form.2:

\[-1\] \sum(j ! 1) = (j ! 1 * (j ! 1 + 1)) / 2
\text{(1)} \quad 1 + \sum(j ! 1) + j ! 1 = (2 + j ! 1 + (j ! 1 * j ! 1 + 2 * j ! 1)) / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D

Run time = 5.62 secs.
Real time = 58.95 secs.

Status

- Type M-x status-proof-theory (M-x spt) and you will see a buffer which displays the formulas in sum (including the TCCs), along with an indication of their proof status
  - This command is useful to see which formulas and TCCs still require proofs
- Another useful command is M-x status-proofchain (M-x spc), which analyzes a given proof to determine its dependencies
Generating LATEX

- Type M-x latex-theory-view (M-x ltv). You will be prompted for the theory name — type sum, or just Return if sum is the default.
- After a few moments the previewer will pop up displaying the sum theory, as shown below.

Generating LATEX (cont’)

```
sum: THEORY
BEGIN
n: VAR nat
sum(n): RECURSIVE nat =
  (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
  MEASURE (λ n : n)
closed_form: THEOREM sum(n) = (n * (n + 1)) / 2
END sum
```
Generating LATEX (cont’)

- Finally using the M-x latex-proof command, it is possible to generate a LATEX file from a proof

Expanding the definition of sum closeed_form.2:

\[ \sum_{i=0}^{j} i = (j \times (j+1))/2 \]

\[ \text{(IF } j' = 1 = 0 \text{ THEN } 0 \text{ ELSE } j' + 1 + \sum_{j''=0}^{j'} \text{ ENDIF)} = ((j'+1) \times (j'+1+1))/2 \]

Conclusions

Advantages of Theorem Proving

- High abstraction and expressive notation
- Powerful logic and reasoning, e.g., induction
- Can exploit hierarchy and regularity, puts user in control
- Can be customized with tactics (programs that build larger proofs steps from basic ones)
- Useful for specifying and verifying parameterized (generic) datapath-dominated designs
- Unrestricted applications (at least theoretically)
Conclusions

Limitations of Theorem Proving:
- Interactive (under user guidance): use many lemmas, large numbers of commands
- Large human investment to prove small theorems
- Usable only by experts: difficult to prove large / hard theorems
- Requires deep understanding of the both the design and HOL (while-box verification)
- must develop proficiency in proving by working on simple but similar problems.
- Automated for narrow classes of designs