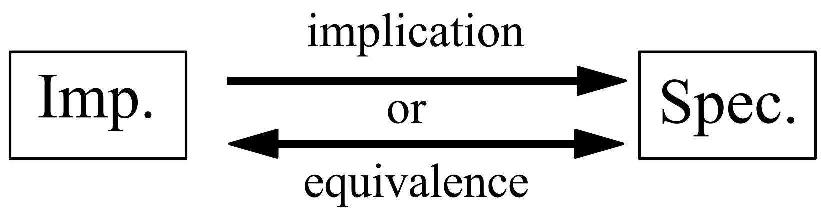
4. Verification by Theorem Proving

Introduction

Theorem Proving

Prove that an implementation satisfies a specification by mathematical reasoning



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.)

Introduction (cont’d)

Proof checking

purported proof → Proof checker “is this a proof?” → “yes”/“no”

- 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

purported theorem → Proof generator “prove this theorem” → a proof

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

First-Order Logic

- *Propositional logic*: reasoning about complete sentences.
- *First-order logic*: also reasoning about *individual objects* and *relationships between them*.

Syntax

- **Objects** (in FOL) are denoted by expressions called ***terms***:

  Constants a, b, c,... ; *Variables* u, v, w,... ;

  $f(t_1, t_2,..., t_n)$ where $t_1, t_2,..., t_n$ are terms and f a *function symbol* of n arguments

- **Predicates**:

  true (T) and *false* (F)

  $p(t_1, t_2,..., t_n)$ where $t_1, t_2,..., t_n$ are *terms* and p a *predicate symbol* of n arguments

- **Formulas**:

  Predicates

  P and Q formulas, then $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, $P \leftrightarrow Q$ are formulas

  x a variable, P a formula, then $\forall x.P$, $\exists x.Q$ are formulas (x is not free in P, Q)

First-Order Logic (cont'd)

Semantics of a first-order logic formulae G: interpretation for function, constant and predicate symbols in G and assigning values to free variables

First-Order Interpretations (Structures) M: M = (D, I)

- D is a non-empty domain of the structure
- I is an interpretation function, assigns function, constant and predicate symbols:
  - (1) For every function symbol f of rank $n>0$, $I(f): D^n \rightarrow D$ is an n-ary function.
  - (2) For every constant c, I(c) is an element of D.
  - (3) For every predicate symbol P of rank $n \geq 0$, $I(P): D^n \rightarrow \{F, T\}$ is an n-ary predicate.

Evaluation

- For every M, a formula can be evaluated to T or F according to the following rules:
  - (1) Evaluate truth values of formulas P and Q, and then the truth values of $\neg P$, $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, $P \leftrightarrow Q$ using propositional logic
  - (2) $\forall x.\ P$ evaluates to T if truth value of G is T for every $d \in D$; otherwise, it is F
  - (3) $\exists x.P$ evaluates to T if truth value of G is T for at least one $d \in D$; otherwise, it is F

First-Order Logic (cont'd)

Example: $G = \forall x.\ (P(x) \rightarrow Q(f(x), a))$, with $M=(D, I)$, $D=\{1,2\}$, and I as:

Assignment for a

a
1

Assignment for f

f(1)	f(2)
2	1

Assignment for P and Q

P(1)	P(2)	Q(1,1)	Q(1,2)	Q(2,1)	Q(2,2)
F	T	T	T	F	T

- $x=1$: $P(x) \rightarrow Q(f(x), a) = P(1) \rightarrow Q(f(1), a) = P(1) \rightarrow Q(2, 1) = F \rightarrow F = T$;
- $x=2$: $P(x) \rightarrow Q(f(x), a) = P(2) \rightarrow Q(f(2), a) = P(2) \rightarrow Q(1, 1) = T \rightarrow T = T$.
- Since $P(x) \rightarrow Q(f(x), a)$ is true for all $x \in D$, $\forall x.\ (P(x) \rightarrow Q(f(x), a))$ is true under M
- M is a model of G ($M \models G$)

(we can also prove that $\exists x.\ (P(x) \rightarrow Q(f(x), a))$ is true under M)

First-Order Logic (cont'd)

The Validity Problem of FOL

- To decide the validity for formulas of FOL, the truth table method does not work!
- *Reason*: must deal with structures not just truth assignments.
- Structures need not be finite ...

Semi-decidable (partially solvable)

- There is an algorithm which starts with an input, and

  1) if the input is valid then
     it terminates after a finite number of steps, and
     outputs the correct value (Yes or No)

  2) if the input is not valid then it reaches a reject halt or loops forever

Theorem (Church-Turing, 1936)

The validity problem for formulas of FOL is undecidable, but semi-decidable.

- Some subsets of FOL are decidable.

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*

Example 1: (mathematical induction)

$\forall P. [P(0) \wedge (\forall n. P(n) \longrightarrow P(n+1))] \longrightarrow \forall n.P(n)$ (Impossible to express it in FOL)

Example 2: Function Rise defined as Rise (c, t) = $\neg c(t) \wedge c(t+1)$

Rise expresses the notion that a signal *c* rises at time *t*.

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

Result of applying Rise to c is a function: $N \rightarrow \{F,T\}$.

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.
- 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 $P(P) = \neg P(P)$,
  (P: bool $\rightarrow$ bool, Q: bool $\rightarrow$ bool) contradiction!

- 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)

Boyer-Moore (Nqthm)

- Developed at University of Texas and later CLI
- Quantifier-free first-order logic.
- Powerful built-in heuristics; user must find a sequence of lemmas that permits to prove the desired theorem with available heuristics
- Collection of LISP programs that permit the user to axiomatize inductively constructed data types, define recursive functions, and (inductively) prove theorems about them
- Process of proof generation is not fully automatic; user assistance for setting up intermediate lemmas and definitions
- A number of verification application including microprocessors
- Tool does not exist anymore!

ACL2

- Developed at CLI
- ACL2 is a mathematical logic together with a mechanical theorem prover to help reason in the logic
- The logic is just a subset of applicative Common Lisp
- The theorem prover is an “industrial strength” version of the Boyer-Moore theorem prover, Nqthm
- Models of all kinds of computing systems can be built in ACL2, just as in Nqthm, even though the formal logic is Lisp
- Once built, an ACL2 model of a system can be *executed* in Common Lisp
- ACL2 can also be used to prove theorems about the model
- For more information: `http://www.cs.utexas.edu/users/moore/acl2/`

PVS

- PVS (Prototype Verification System) developed at SRI
- The specification language of PVS is based on classical, typed higher-order logic
- The primitive inferences include propositional and quantifier rules, induction, rewriting, and *decision procedures* for linear arithmetic
- The implementations of these primitive inferences are optimized for large proofs: E.g., propositional simplification uses BDDs, and auto-rewrites are cached for efficiency
- User-defined procedures can combine these primitive inferences to yield higher-level proof strategies
- PVS includes a *BDD-based decision procedure* for relational Mu-calculus: experimental integration of theorem proving and CTL model checking
- Proofs are developed interactively by combining high-level inference procedures:
- For more information: `http://pvs.csl.sri.com/`

Lambda

- Commercial tool by Abstract Hardware Ltd. (UK)
- Verification and synthesis tool based on high-order logic theorem proving
- Specification in predicate logic and expressed in the L2 language (based on SML, Standard Meta Language)
- Specification can be executed using the "Animator" tool
- Interactive *correct-by-construction* synthesis using
  - transformations by applying rewriting rules
  - partitioning
  - instantiating and interconnecting components
  - scheduling operations, and allocating resources (even for pipelined designs)
- Backtracking to a preceding design and exploration of alternatives
- Reasoning over a mix of timing scales, e.g., clock ticks, frame periods, pipeline insertion
- Output current state of the design (subset of L2) in **VHDL** and produce control microcode
- Complex properties can be stated and proven as formulas to be satisfied by the design
- Tool does not exist anymore!

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
- 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
- For more information: `https://hol-theorem-prover.org/`

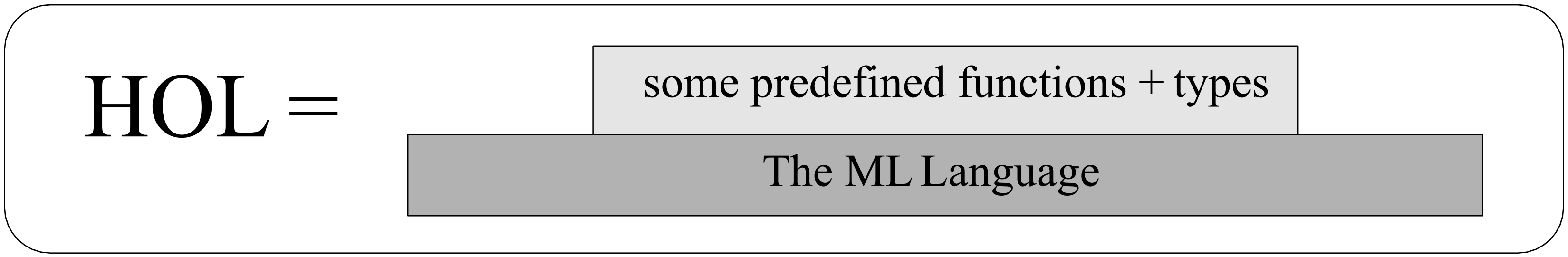
Note: we will now focus on HOL!

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
- 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
  - HOL4: Standard ML

HOL Theorem Prover (cont'd)

- HOL and ML



- 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.)

HOL Theorem Prover (cont'd)

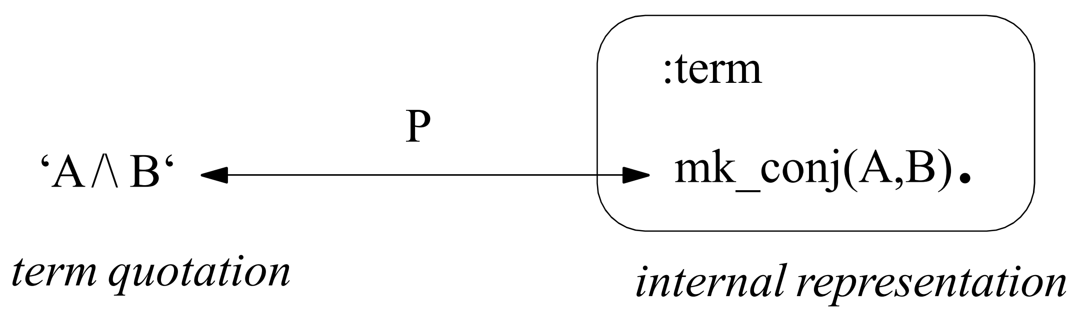
- How the logic is embedded in ML:

logic	terms	types	theorems
ML data type	**:term**	**:hol_type**	**:thm**

- Terms are represented by values of the ML abstract data type **:term**

```
- P `T /\ F ==> T`;
val it = `T/\ F ==> T` : term
```

- The quotation parser and pretyyprinter:



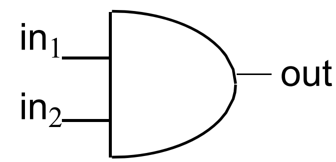
Specification in HOL

- **Functional description**:

  express output signal as function of input signals, e.g.:

  AND gate:

  $$out = \textbf{and}\ (in_1, in_2) = (in_1 \wedge in_2)$$



- **Relational (predicate) description**:

  gives relationship between inputs and outputs in the form of a predicate (a Boolean function returning "true" of "false"), e.g.:

  AND gate:

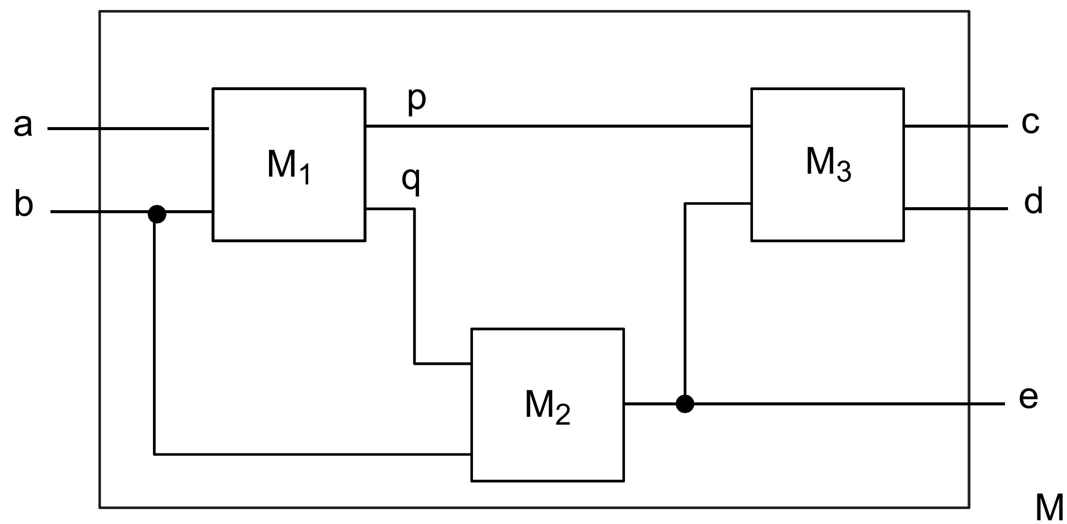
  $$\textbf{AND}\ ((in_1, in_2),(out)) := out = (in_1 \wedge in_2)$$

Notes:

- functional descriptions allow recursive functions to be described. They cannot describe bi-directional signal behavior or functions with multiple feed-back 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

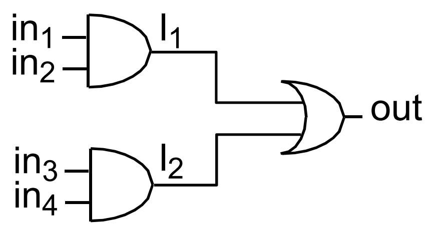


- conjunction "$\wedge$" of implementation module predicates $M(a, b, c, d, e) := M_1(a, b, p, q) \wedge$
  $M_2(q, b, e) \wedge$
  $M_3(e, p, c, d)$
- hide internal lines (p,q) using **existential quantification**

$$M(a, b, c, d, e) := \exists\, p\, q.\; M_1(a, b, p, q) \wedge M_2(q, b, e) \wedge M_3(e, p, c, d)$$

Specification in HOL

Combinational circuits



SPEC $(in_1, in_2, in_3, in_4, out) :=$

$out = (in_1 \wedge in_2) \vee (in_3 \wedge in_4)$

IMPL $(in_1, in_2, in_3, in_4, out) :=$

$\exists\, l_1\, l_2.$ **AND** $(in_1, in_2, l_1) \wedge$ **AND** $(in_3, in_4, l_2) \wedge$ **OR** (l_1, l_2, out)

where **AND** $(a, b, c) := (c = a \wedge b)$

OR $(a, b, c) := (c = a \vee b)$

Note: a functional description would be:

IMPL $(in_1, in_2, in_3, in_4, out) :=$

$out =$ (or (and (in_1, in_2), and (in_3, in_4)))

where **and** $(in_1, in_2) = (in_1 \wedge in_2)$

or $(in_1, in_2) = (in_1 \vee in_2)$

Specification in HOL

Sequential circuits

- Explicit expression of time (discrete time modeled as natural numbers).
- Signals defined as functions over time, e.g. type: (**nat** $\rightarrow$ **bool**) or (**nat** $\rightarrow$ **bitvec**)
- Example: D-flip-flop (latch):

  DFF (in, out):= (out (0) = F) $\wedge$ ($\forall$ t. out (t+1) = in (t))

  in and *out* are functions of time *t* to boolean values: type (nat $\rightarrow$ bool)

- Notion of time can be added to combinational circuits, e.g., AND gate

  AND (in_1, in_2, out):= $\forall$ t. out (t) = (in_1(t) $\wedge$ in_2(t))

- Temporal operators can be defines as predicates, e.g.:

  EVENTUAL sig t_1 = $\exists$ t_2. ($t_2 > t_1$) $\wedge$ sig t_2

  meaning that signal "sig" will eventually be true at time $t_2 > t_1$.

  Note: This kind of specification using existential quantified time variables is useful to describe asynchronous behavior

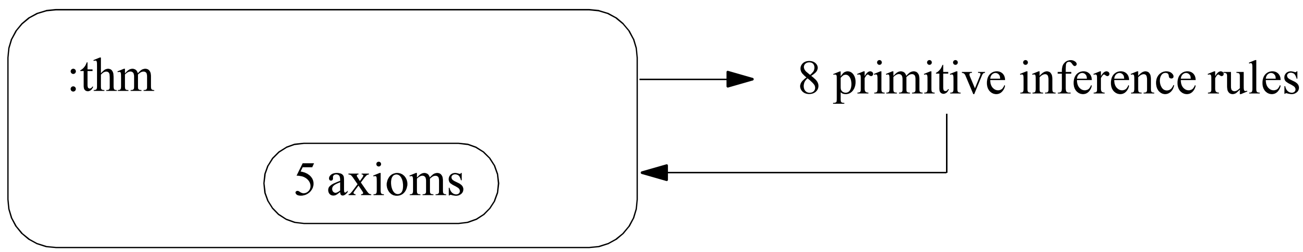
HOL Proof Mechanism

- 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 Γ 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:

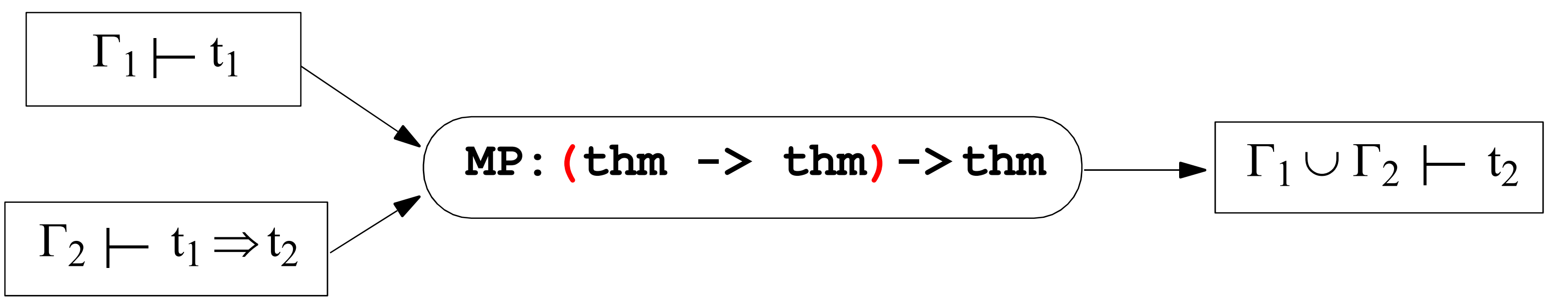
  All theorems must be proved!

HOL Proof System

- In the core of HOL:



- An *inference rule*: as an ML function that returns a theorem as a result
- Example: *modus ponens* in HOL,



- The function returns only objects of type **`thm`** that logically follow by the inference rule

Primitive Rules

- All theorems in HOL are ultimately proved using only the primitive inference rule:

ASSUME:

$$\frac{}{\{t\} \vdash t}$$

REFL:

$$\frac{}{\vdash t = t}$$

BETA CONVERSION:

$$\frac{}{\vdash (\lambda x.\ t_1)\ t_2 = t_1[t_2/x]}$$

ABSTRACTION:

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)} \text{ (x not free in } \Gamma\text{)}$$

INST_TYPE:

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

DISCHARGE:

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}$$

MODUS PONENS:

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2\ ,\ \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

SUBST:

$$\frac{\Gamma_1 \vdash t_1 = t_1',\ \ldots..,\ \Gamma_n \vdash t_n = t_n',\ \Gamma \vdash t}{\Gamma_1 \cup \ldots.. \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n/t_1', \ldots, t_n']}$$

Basic Rewriting Rules

- Rewriting is done:
  - with all the supplied equations
  - on all subterms of the theorem to be rewritten
  - repeatedly, until no rewrite rule applies

- Rewriting rules:

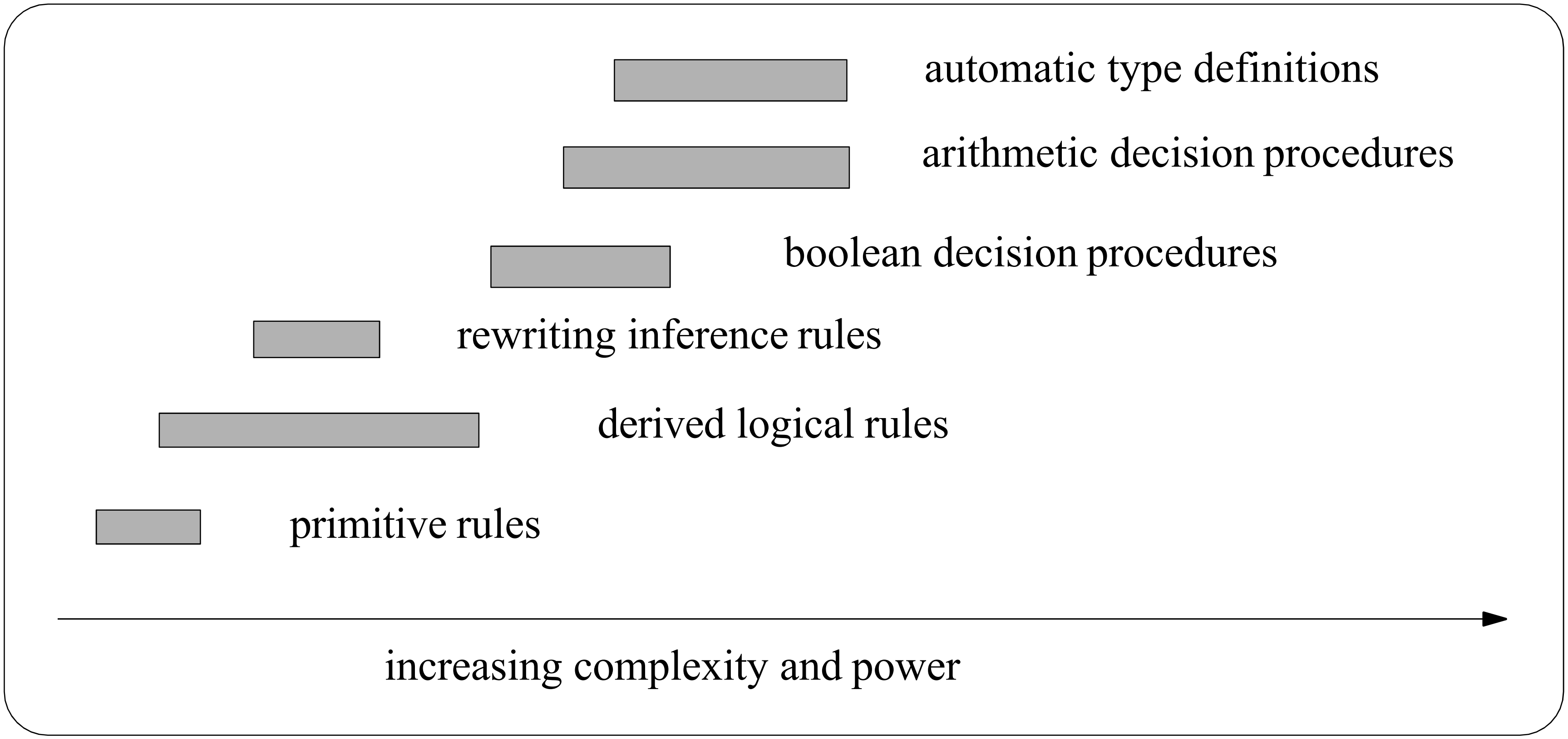
Rewriting rules: `thm list -> thm -> thm`

theorems to be rewritten

list of equational theorems to be used as left-to-right rewrite rules

Built-in Derived Rules

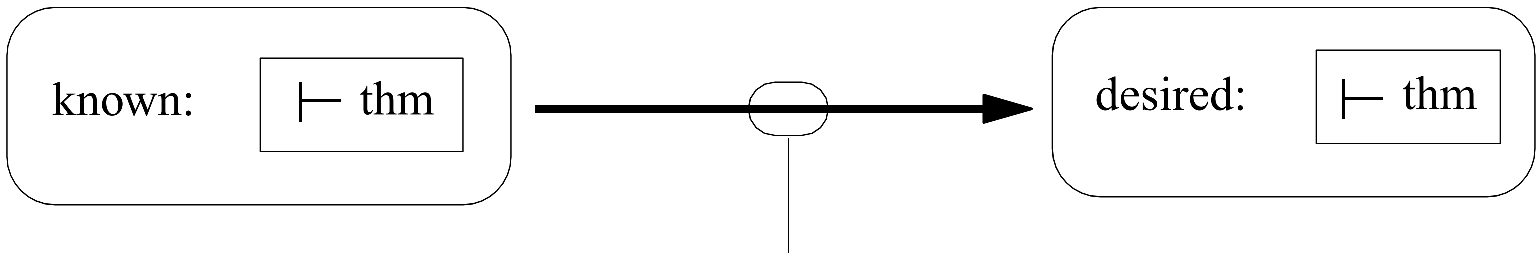
- There is a wide range of derived inference rules built into the system:



- To become an expert HOL user, one should continuously learn new rules and proof techniques

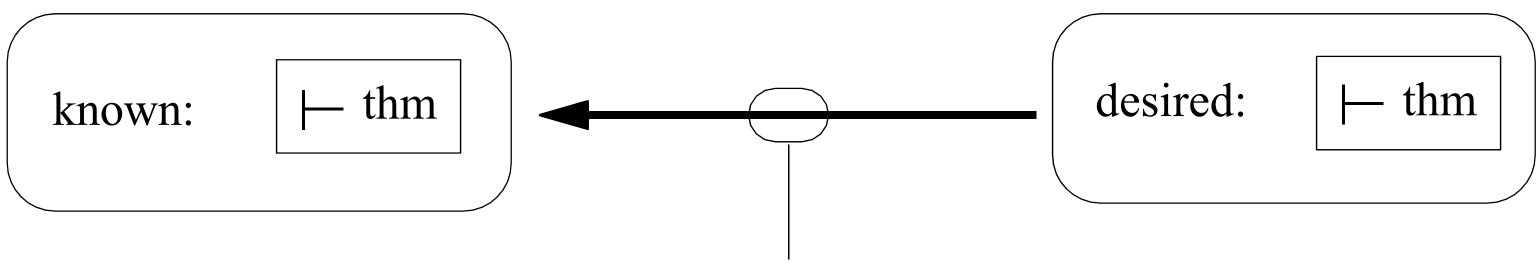
Proof Styles in HOL

- **Forward** proof style:



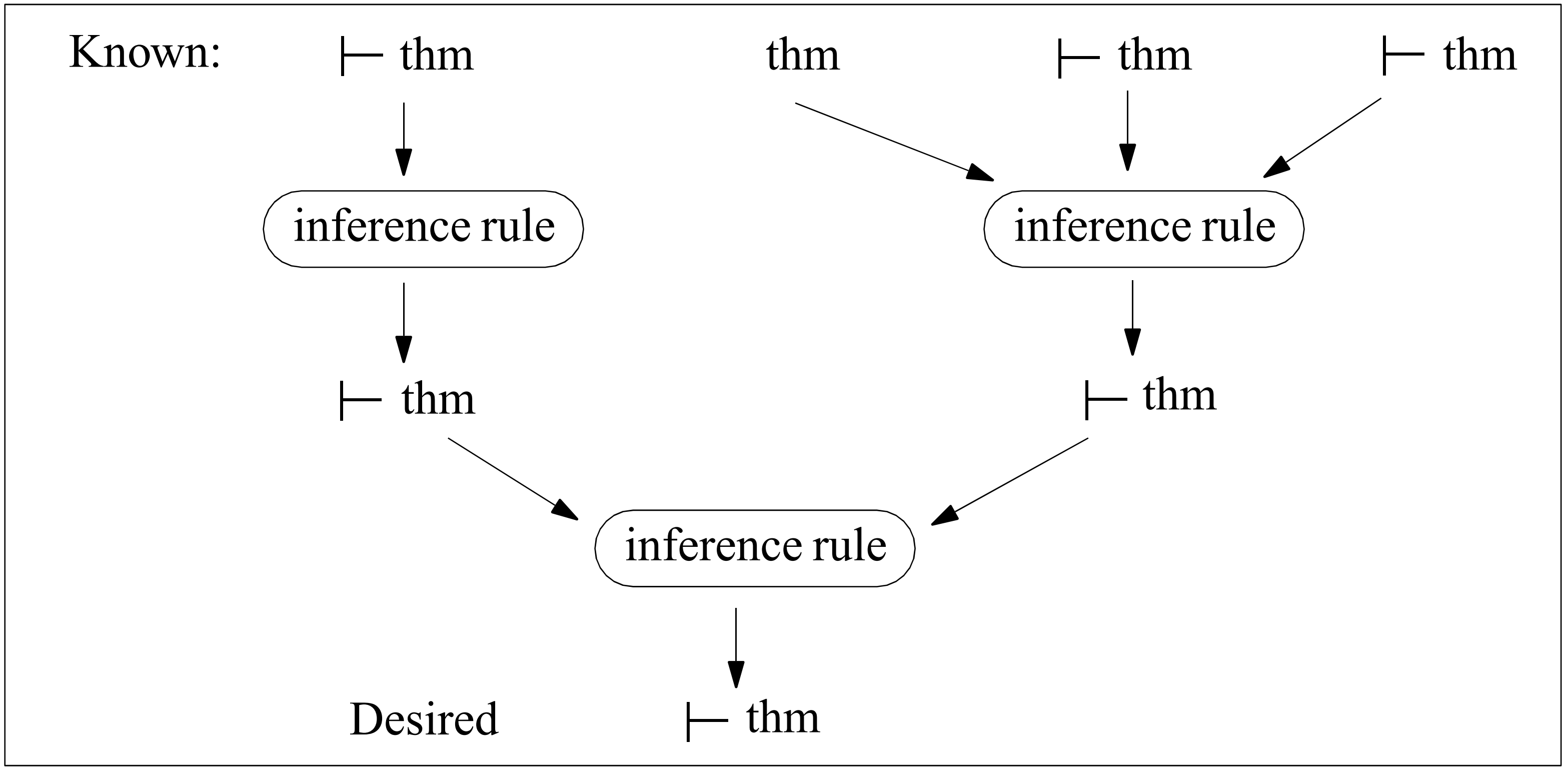
apply inference rules to infer the desired theorem from already proved theorems

- Goal-directed (or **Backward**) proof style:



reduce goal to be proved to simpler and simpler subgoals until known theorems reached

Forward Proof in HOL



- can be millions of (primitive) inferences long
- usually not natural for “one-off” proofs
- but essential for tool building

Backward Proof in HOL

- Goals are represented by values of ML type

goal: term list $*$ term

assumptions — conclusion

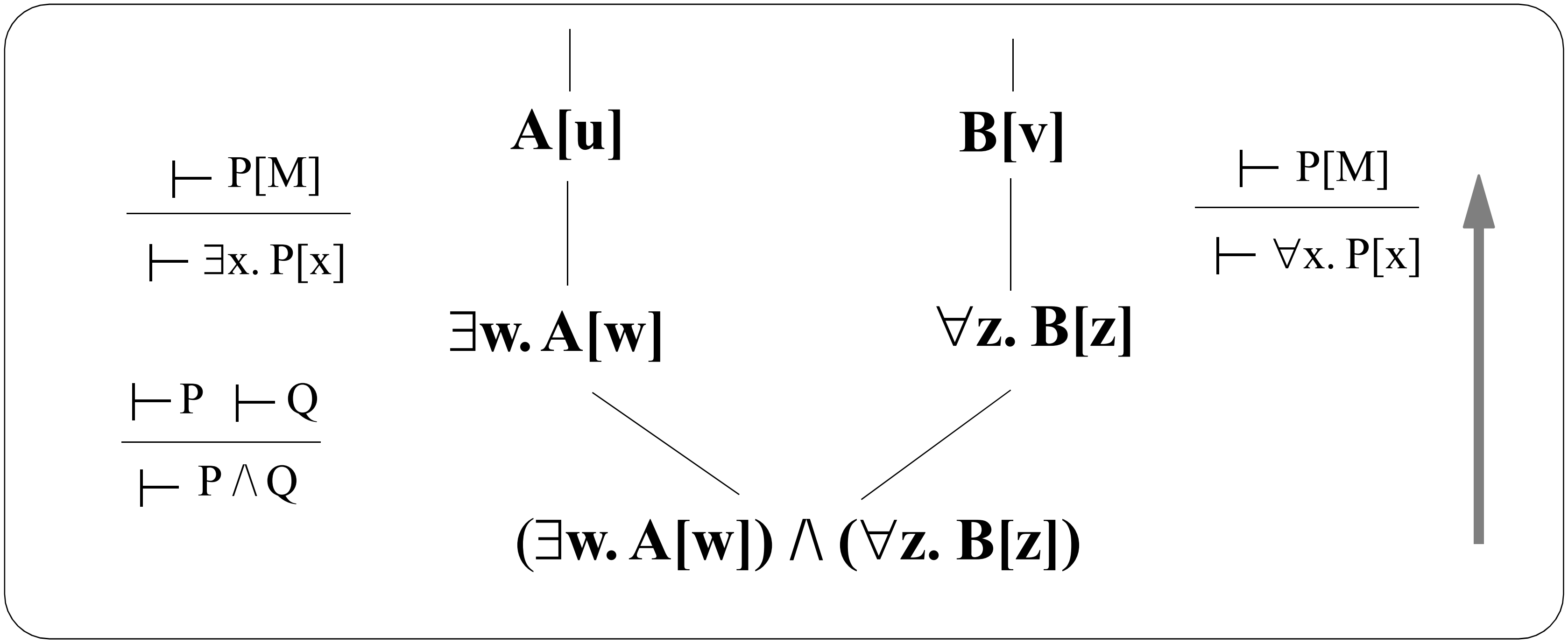
$$\underset{:\ \mathrm{thm}}{P_1, \ldots, P_n \vdash P} \xrightarrow{\text{solves}} \underset{:\ \mathrm{goal}}{([P_1, \ldots, P_n], P)}$$

- Goal-directed proof in HOL:

$$\begin{tikzcd}
\text{|- thm} \arrow[d, equal] & & \vdash \text{goal} \arrow[ll, "\textit{user}\text{ works backwards}"'] \\
\text{|- thm} \arrow[rr, "\textit{machine}\text{ builds the proof}"] & & \vdash \text{thm} \arrow[u, "\text{solves}"']
\end{tikzcd}$$

Backward Proof

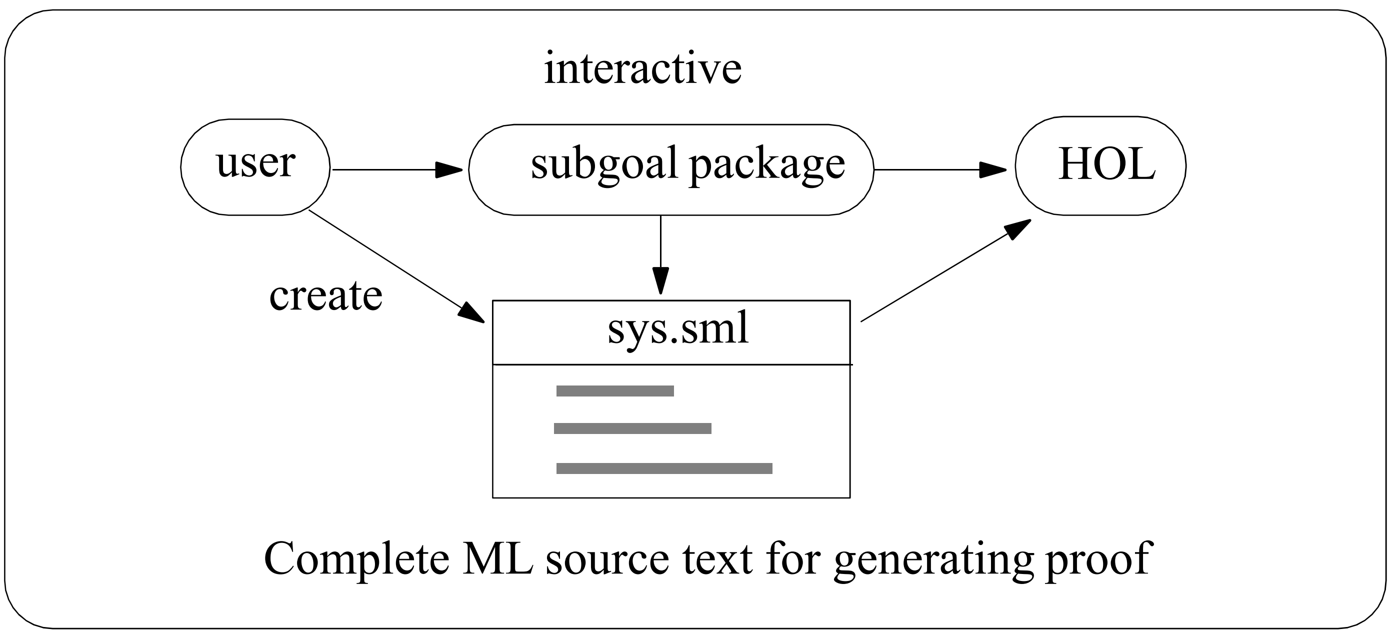
- Example:



- Reduction of a goal to *subgoals* is justified by an inference in the “opposite direction”.

The Subgoal Package

- HOL has a subgoal package for finding tactic proofs interactively
- The subgoal package:
  - maintains a *stack* of subgoals to be proved
  - provides functions that operate on these subgoals
- The subgoal package is for finding the schema of the proof:



HOL Tactics

- Tactic is a function:

$$T : \texttt{goal -> } \underbrace{\texttt{goal list}}_{\text{subgoals}} \; \underbrace{\texttt{(thm list -> thm)}}_{\text{justification}}$$

- Suppose that for a given goal g: $T(g) = ([g_1, ..., g_n], f)$
- If the theorems $\Gamma_1 \vdash P_1, ..., \Gamma_n \vdash P_n$ solve the goals $g_1, ..., g_n$, then $f([\Gamma_1 \vdash P_1, ..., \Gamma_n \vdash P_n])$ should solve the original goal g.
- In a picture:

$$\begin{tikzcd}
([g_1, ..., g_n], f) \arrow[d, equal] & & g \arrow[ll, "\text{Tactics}"'] \\
{[\Gamma_1 \vdash P_1, ..., \Gamma_n \vdash P_n]} \arrow[rr, "f"] & & \Gamma \vdash P \arrow[u, "\text{solves}"']
\end{tikzcd}$$

HOL Tactics (Examples)

$$\frac{A \vdash g}{A \cup \{t\} \vdash g} \quad \text{ASSUM\_TAC } (A \vdash t)$$

$$\frac{A \vee B}{A} \quad \text{DISJ1\_TAC} \qquad \frac{A \vee B}{B} \quad \text{DISJ2\_TAC}$$

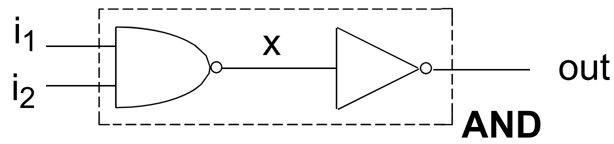
$$\frac{A \vdash t_1 = t_2}{A \vdash t_1 \Rightarrow t_2 \qquad A \vdash t_2 \Rightarrow t_1} \quad \text{EQ\_TAC}$$

$$\frac{A \vdash \forall x.\, P}{A \vdash P[x'/x]} \quad \text{GEN\_TAC}$$

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

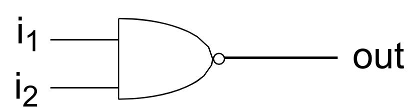


AND Specification:

$AND\_SPEC\ (i_1,i_2,out) := out = i_1 \wedge i_2$

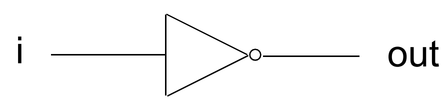
NAND specification:

$NAND\ (i_1,i_2,out) := out = \neg(i_1 \wedge i_2)$



NOT specification:

$NOT\ (i, out) := out = \neg i$



AND Implementation:

$AND\_IMPL\ (i_1,i_2,out) := \exists x.\ NAND\ (i_1,i_2,x) \wedge NOT\ (x,out)$

Logic AND (cont'd)

Proof Goal:

$\forall$ i_1, i_2, out. AND_IMPL (i_1,i_2,out) $\Rightarrow$ AND_SPEC (i_1,i_2,out)

Proof (forward)

AND_IMP (i_1,i_2,out) {from above circuit diagram}

$\vdash$ $\exists$x.NAND (i_1,i_2,x) $\wedge$ NOT (x,out) {by def. of AND_IMP}

$\vdash$ NAND (i_1,i_2,***x***) $\wedge$ NOT(***x***,out) {strip off "$\exists$x."}

$\vdash$ NAND (i_1,i_2,***x***) {left conjunct of line 3} x

$\vdash$ x = $\neg$($i_1$$\wedge$$i_2$) {by def. of NAND}

$\vdash$ NOT (*x*,out) {right conjunct of line 3}

$\vdash$ out = $\neg$x {by def. of NOT}

$\vdash$ out = $\neg$($\neg$($i_1$$\wedge$$i_2$) {substitution, line 5 into 7}

$\vdash$ out = ($i_1$$\wedge$$i_2$) {simplify,$\neg\neg$ t=t}

$\vdash$ AND (i_1,i_2,out) {by def. of AND spec}

$\vdash$ AND_IMPL (i_1,i_2,out) $\Rightarrow$ AND_SPEC (i_1,i_2,out)

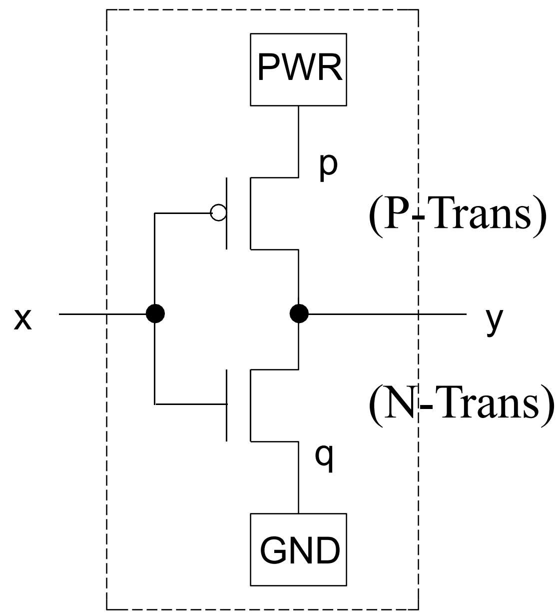
Q.E.D.

Example 2: CMOS-Inverter

Specification (black-box behavior)

SPEC(x,y):= (y = ¬x)

Implementation



Basic Modules Specs

PWR(x):= (x = T)

GND(x):= (x = F)

N-Trans(g,x,y):= (g ⇒ (x = y))

P-Trans(g,x,y):= (¬ g ⇒ (x = y))

Implementation (network structure)

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := \exists\, p\, q.\ & \mathrm{PWR}(p) \wedge \\
& \mathrm{GND}(q) \wedge \\
& \text{N-Tran}(x,y,q) \wedge \\
& \text{P-Tran}(x,p,y)
\end{aligned}
$$

Proof goal

$$\forall\, x\ y.\ \mathrm{IMPL}(x,y) \Leftrightarrow \mathrm{SEPC}(x,y)$$

Proof (forward)

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := \exists\, p\, q.\ & (p = T) \wedge \\
& (q = F) \wedge \qquad \text{(substitution of the definition of PWR and GND)} \\
& \text{N-Tran}(x,y,q) \wedge \\
& \text{P-Tran}(x,p,y)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := \exists\, p\, q.\ & (p = T) \wedge \\
& (q = F) \wedge \qquad \text{(substitution of p and q in P-Tran and N-Tran)} \\
& \text{N-Tran}(x,y,F) \wedge \\
& \text{P-Tran}(x,T,y)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := {} & (\exists\, p.\ p = T) \wedge \\
& (\exists\, q.\ q = F) \wedge \\
& \text{N-Tran}(x,y,F) \wedge \\
& \text{P-Tran}(x,T,y)
\end{aligned}
$$

(use Thm: "$\exists$ a. t1 $\wedge$ t2= ($\exists$ a. t_1) $\wedge$ t_2" if a is free in t_2)

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := {} & T \wedge \\
& T \wedge \\
& \text{N-Tran}(x,y,F) \wedge \\
& \text{P-Tran}(x,T,y)
\end{aligned}
$$

(use Thm: "($\exists$ a. a=T) = T" and "($\exists$ a. a=F) = T")

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := {} & \text{N-Tran}(x,y,F) \wedge \\
& \text{P-Tran}(x,T,q)
\end{aligned}
$$

(use Thm: "$x \wedge T = x$")

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := {} & (x \Rightarrow (y = F)) \wedge \\
& (\neg x \Rightarrow (T = y))
\end{aligned}
$$

(use def. of N-Tran and P-Tran)

$$
\begin{aligned}
\mathrm{IMPL}(x,y) := {} & (\neg x \vee (y = F)) \wedge \\
& (x \vee (T = y))
\end{aligned}
$$

(use "$(a \Rightarrow b) = (\neg a \vee b)$")

Boolean simplifications:

$$IMPL(x,y) := (\neg x \wedge x) \vee (\neg x \wedge (T = y)) \vee ((y = F) \wedge x) \vee ((y = F) \wedge (T = y))$$

$$IMPL(x,y) := F \vee (\neg x \wedge (T = y)) \vee ((y = F) \wedge x) \vee F$$

$$IMPL(x,y) := (\neg x \wedge (T = y)) \vee ((y = F) \wedge x)$$

Case analysis x=T/F

$$x=T: IMPL(T,y) := (F \wedge (T = y)) \vee ((y = F) \wedge T)$$

$$x=F: IMPL(F,y) := (T \wedge (T = y)) \vee ((y = F) \wedge F)$$

$$x=T: IMPL(T,y) := (y = F)$$

$$x=F: IMPL(F,y) := (T = y)$$

$$\stackrel{!}{=}$$

Case analysis on **SPEC**:

$$x=T: SPEC(T,y) := (y=F)$$

$$x=F: SPEC(F,y) := (y=T)$$

Conclusion: $\vdash SPEC(x,y) \Leftrightarrow IMPL(x,y)$

Abstraction Forms

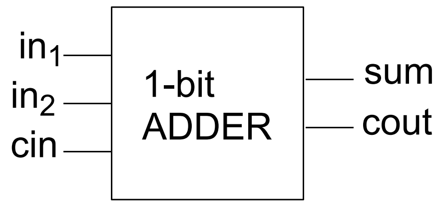
- **Structural abstraction:** only the behavior of the external inputs and outputs of a module is of interest (abstracts away any internal details)

- **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)

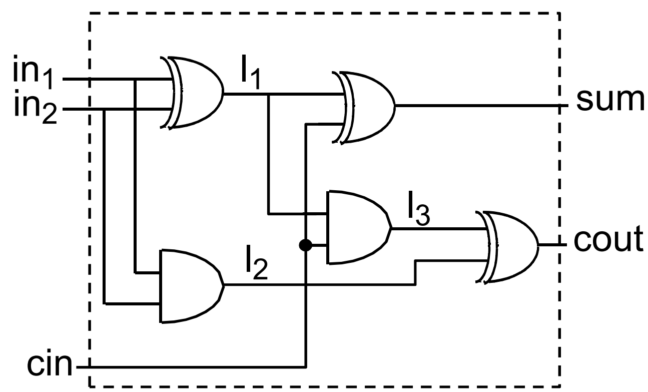
Example 3: 1-bit Adder



Specification:

ADDER_SPEC (in_1:**nat**, in_2:**nat**, cin:**nat**, sum:**nat**, cout:**nat**):= $in_1 + in_2 + cin = 2 * cout + sum$

Implementation:



<u>**Note**</u>: Spec is a **behavioral abstraction** of Impl.

1-bit Adder (cont'd)

Implementation:

$$\text{ADDER\_IMPL}\ (in_1{:}\mathbf{bool},\ in_2{:}\mathbf{bool},\ cin{:}\mathbf{bool},\ sum{:}\mathbf{bool},\ cout{:}\mathbf{bool}) :=$$

$$\begin{aligned}\exists\, l_1\, l_2\, l_3.\ & \mathsf{EXOR}\ (in_1, in_2, l_1) \wedge \\ & \mathsf{AND}\ (in_1, in_2, l_2) \wedge \\ & \mathsf{EXOR}\ (l_1, cin, sum) \wedge \\ & \mathsf{AND}\ (l_1, cin, l_3) \wedge \\ & \mathsf{OR}\ (l_2, l_3, cout)\end{aligned}$$

Define a **data abstraction function** (**bn: bool** $\rightarrow$ **nat**) needed to relate Spec variable types (nat) to Impl variable types (bool):

$$bn(x) \coloneqq \begin{cases} 1, & \text{if } x = T \\ 0, & \text{if } x = F \end{cases}$$

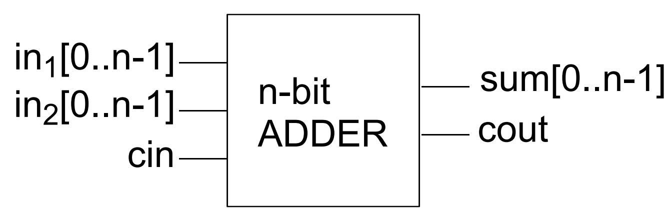
Proof goal:

$$\begin{aligned}&\forall\ in_1, in_2, cin, sum, cout. \\ &\quad \text{ADDER\_IMPL}\ (in_1, in_2, cin, sum, cout) \\ &\quad\quad \Rightarrow \text{ADDER\_SPEC}\ (\mathbf{bn}(in_1), \mathbf{bn}(in_2), \mathbf{bn}(cin), \mathbf{bn}(sum), \mathbf{bn}(cout))\end{aligned}$$

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: N-bit Adder

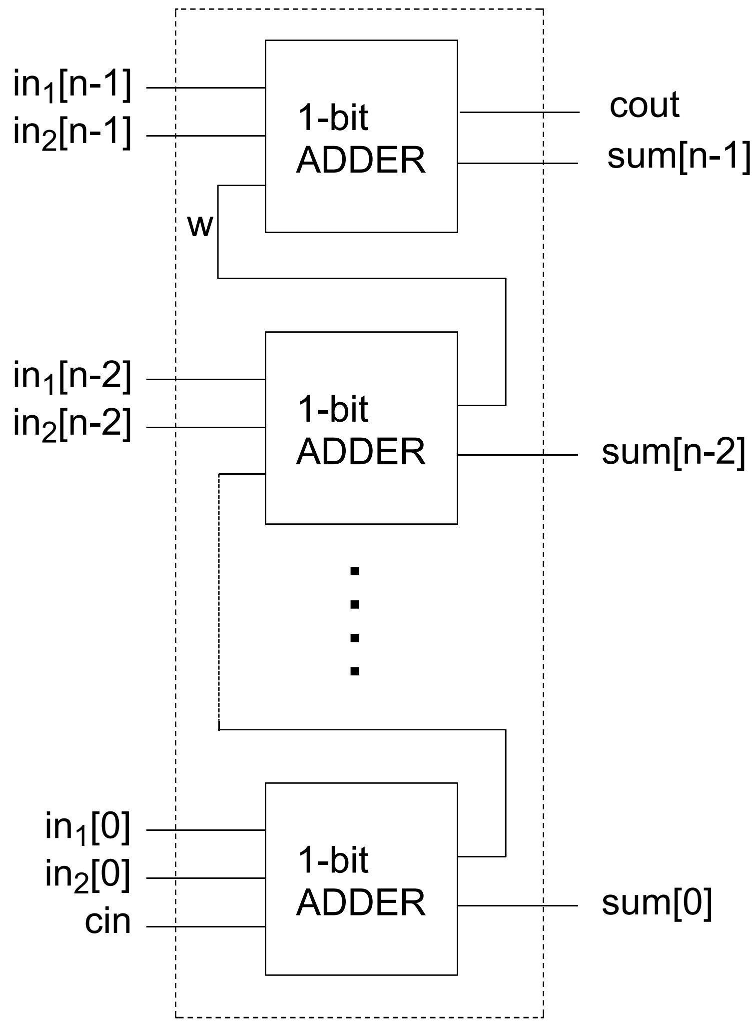


Specification

N-ADDER_SPEC (**n**,in_1,in_2,cin,sum,cout):= (in_1 + in_2 + cin = 2^{n+1} * cout + sum)

Example 4: N-bit Adder

Implementation



N-bit Adder (cont'd)

Implementation

- recursive definition:

  N-ADDER_IMP $(n, in_1[0..n-1], in_2[0..n-1], cin, sum[0..n-1], cout) :=$

  $\exists$ **w**. N-ADDER_IMP $(n-1, in_1[0..n-2], in_2[0..n-2], cin, sum[0..n-2], w) \wedge$

  N-ADDER_IMP $(1, in_1[n-1], in_2[n-1], w, sum[n-1], cout)$

- Note: N-ADDER_IMP $(1, in_1[i], in_2[i], cin, sum[i], cout)$

  $=$ ADDER_IMP $(in_1[i], in_2[i], cin, sum[i], cout)$

- Data abstraction function (**vn: bitvec** $\rightarrow$ **nat**) to relate bit vctors to natural numbers:

  $vn(x[0]) := bn(x[0])$

  $vn(x[0,n]) := 2^n * bn(x[n]) + vn(x[0,n-1]$

Proof goal:

$\forall$ **n**, in_1, in_2, cin, sum, cout.

N-ADDER_IMP $(n, in_1[0..n-1], in_2[0..n-1], cin, sum[0..n-1], cout)$

$\Rightarrow$ N-ADDER_SPEC (n,**vn**(in_1[0..n-1]),**vn**(in_2[0..n-1]),**vn**(cin),**vn**(sum[0..n-1]),**vn**(cout))

can be **instantiated** with **n = 32:**

$\forall$ in_1, in_2, cin, sum, cout.

N-ADDER_IMP $(in_1[0..31], in_2[0..31], cin, sum[0..31], cout)$

$\Rightarrow$ N-ADDER_SPEC (**vn**(in_1[0..31]),**vn**(in_2[0..31]),**vn**(cin),**vn**(sum[0..31]),**vn**(cout))

N-bit Adder (cont'd)

Proof by induction over n:

- basis step:
  N-ADDER_IMP $(0,in_1[0],in_2[0],cin,sum[0],cout)$
  $\Rightarrow$ N-ADDER_SPEC $(0,\mathbf{vn}(in_1[0]),\mathbf{vn}(in_2[0]),\mathbf{vn}(cin),\mathbf{vn}(sum[0]),\mathbf{vn}(cout))$
- induction step:
  $\big[$N-ADDER_IMP $(n,in_1[0..n-1],in_2[0..n-1],cin,sum[0..n-1],cout) \Rightarrow$
  N-ADDER_SPEC $(n,\mathbf{vn}(in_1[0..n-1]),\mathbf{vn}(in_2[0..n-1]),\mathbf{vn}(cin),\mathbf{vn}(sum[0..n-1]),\mathbf{vn}(cout))\big]$
  $\Rightarrow$
  $\big[$N-ADDER_IMP$(n+1,in_1[0..n],in_2[0..n],cin,sum[0..n],cout) \Rightarrow$
  N-ADDER_SPEC$(n+1,\mathbf{vn}(in_1[0..n]),\mathbf{vn}(in_2[0..n]),\mathbf{vn}(cin),\mathbf{vn}(sum[0..n]),\mathbf{vn}(cout))\big]$

Notes:

- basis step is equivalent to 1-bit adder proof, i.e.
  ADDER_IMP $(in_1[0],in_2[0],cin,sum[0],cout)$
  $\Rightarrow$ ADDER_SPEC $(\mathbf{bn}(in_1[0]),\mathbf{bn}(in_2[0]),\mathbf{bn}(cin),\mathbf{bn}(sum[0]),\mathbf{bn}(cout))$
- 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)
- 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

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)

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

References

Logic:

1. Fitting, M.: *First-Order Logic and Automated Theorem Proving*; Springer-Verlag, 1990.
2. Andrews, P.: *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*; Academic Press, 1986.

HOL:

3. Gordon, M.; Melham, T.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*; Cambridge University Press, 1993.
4. Melham, T.F. and Gordon, M.J.C.: *Higher Order Logic and Hardware Verification*; Cambridge University Press, 1993.

ACL2:

6. Boyer, R.; Moore, J.: *A Computational Logic Handbook*; Academic Press, 1988.

PVS:

7. Owre, S.; Shankar, N.; Rushby, J.: User Guide for the PVS Specification and Verification System, Language, and Proof Checker; Computer Science Laboratory, SRI International, Melno Park, California, February 1993.