

Formal Specification and Verification

Classical logic (4)

6.11.2018

Viorica Sofronie-Stokkermans

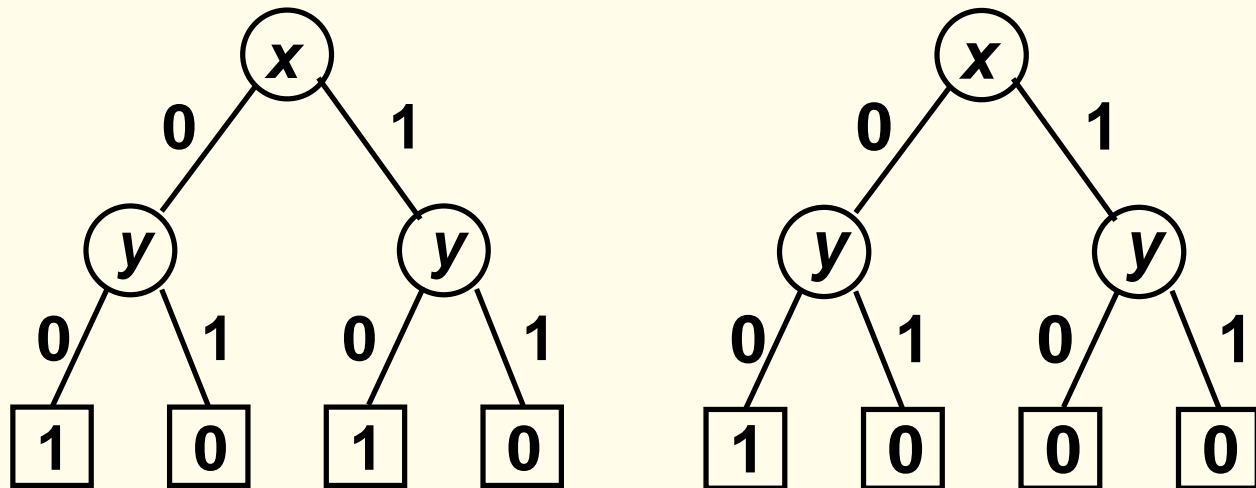
e-mail: sofronie@uni-koblenz.de

Binary Decision Diagrams

Formulae \leftrightarrow Boolean functions

F (n Prop.Var) $\mapsto f_F : \{0, 1\}^n \rightarrow \{0, 1\}$

Binary decision trees:

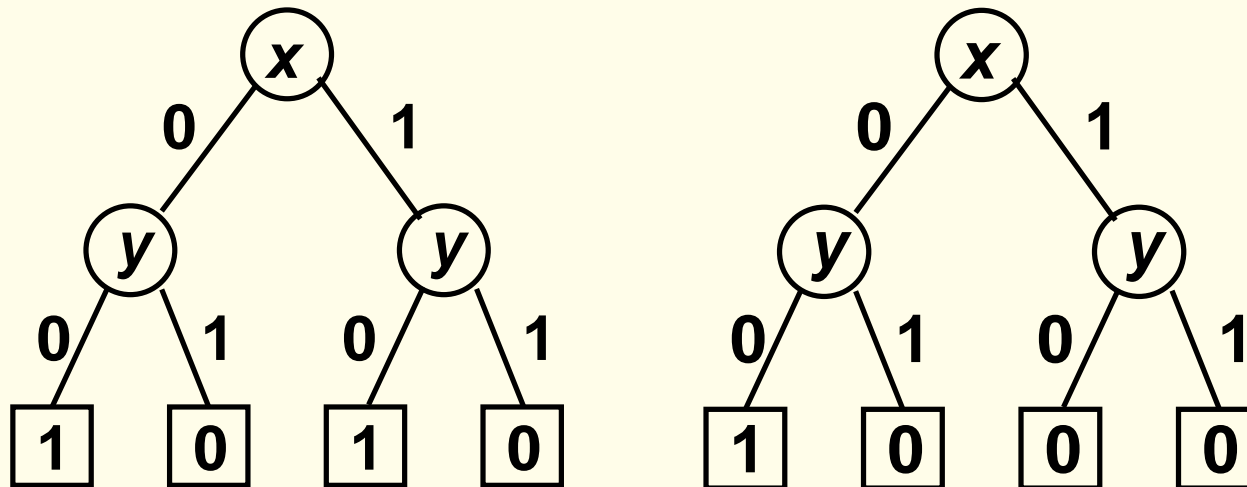


Binary Decision Diagrams

Formulae \leftrightarrow Boolean functions

F (n Prop.Var) $\mapsto f_F : \{0, 1\}^n \rightarrow \{0, 1\}$

Binary decision trees:



- exactly as inefficient as truth tables ($2^{n+1} - 1$ nodes if n prop.vars.)
- optimization possible: remove redundancies

Binary Decision Diagrams

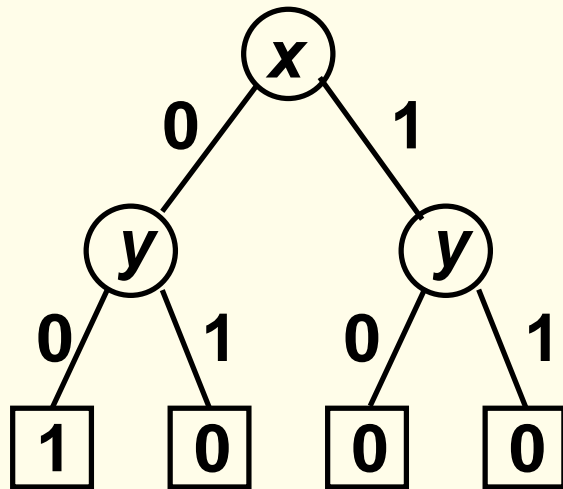
Optimization: remove redundancies

1. remove duplicate leaves
2. remove unnecessary tests
3. remove duplicate nodes

Binary Decision Diagrams

1. remove duplicate leaves

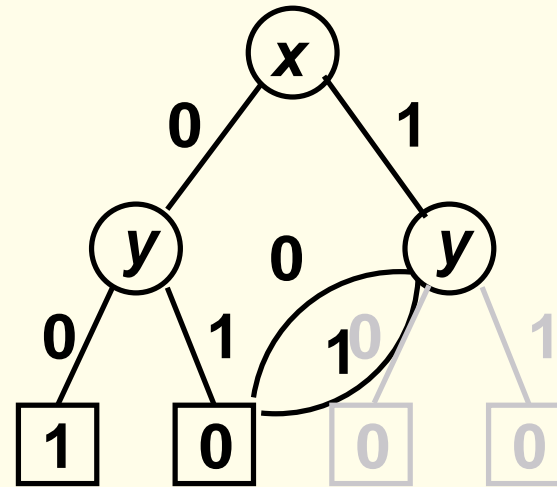
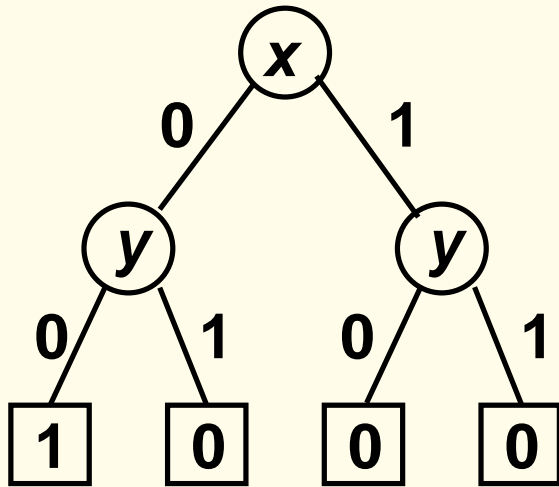
Only one copy of 0 and 1 necessary:



Binary Decision Diagrams

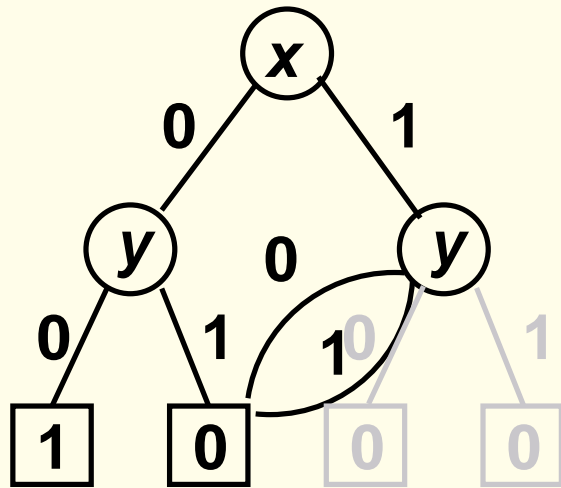
1. remove duplicate leaves

Only one copy of 0 and 1 necessary:



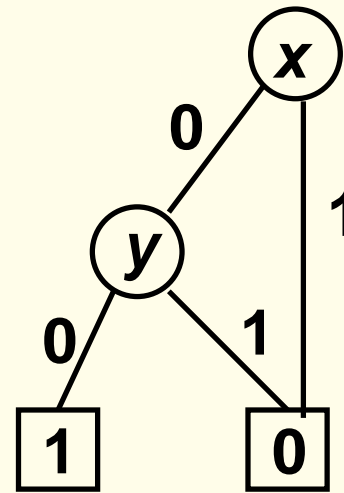
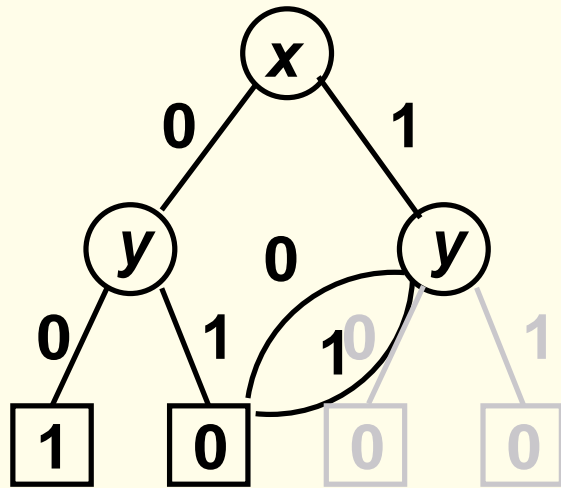
Binary Decision Diagrams

2. remove unnecessary tests



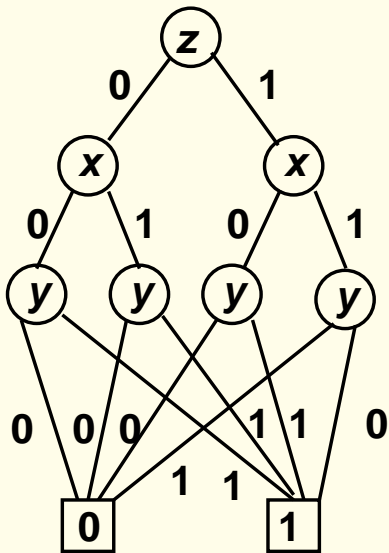
Binary Decision Diagrams

2. remove unnecessary tests



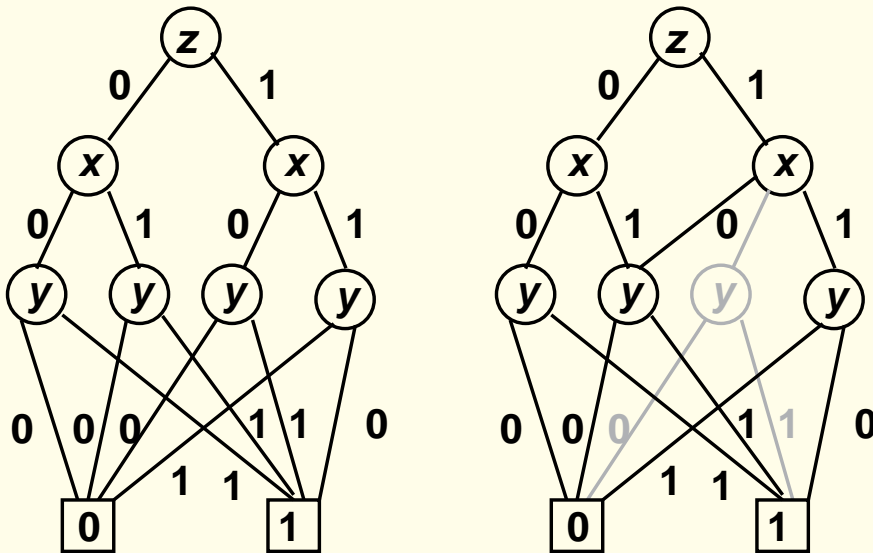
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



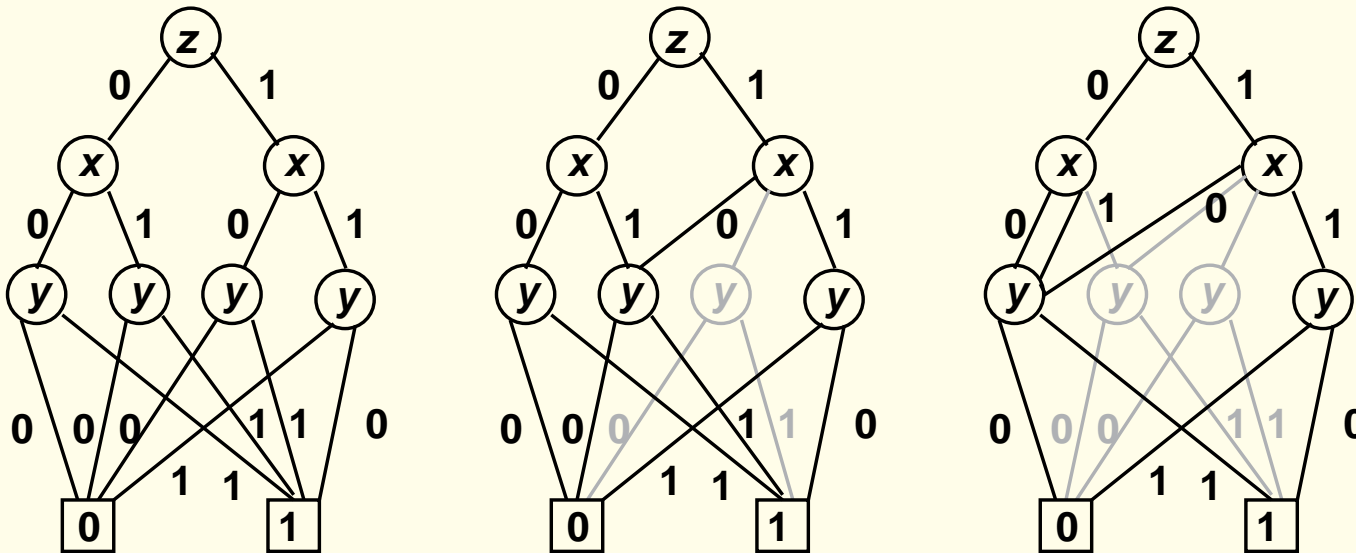
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



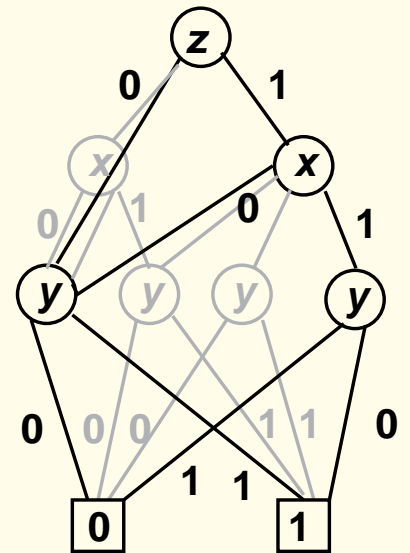
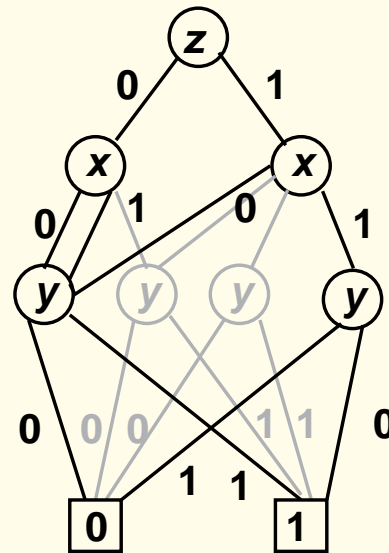
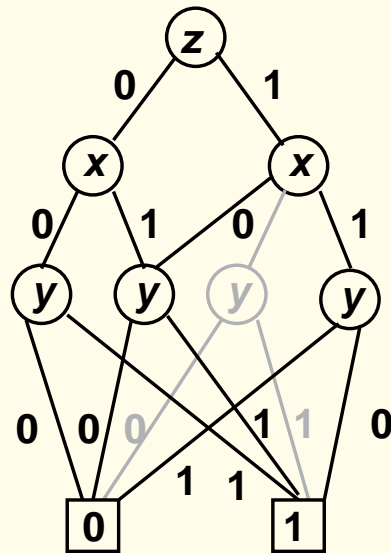
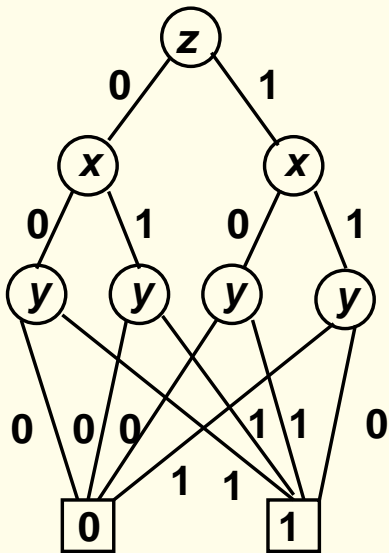
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



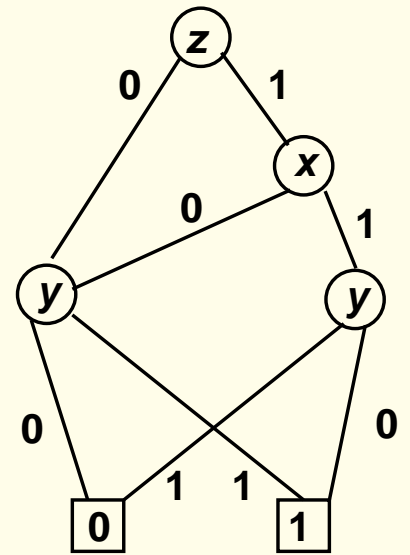
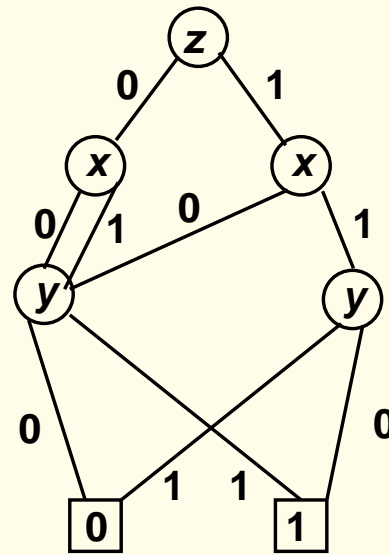
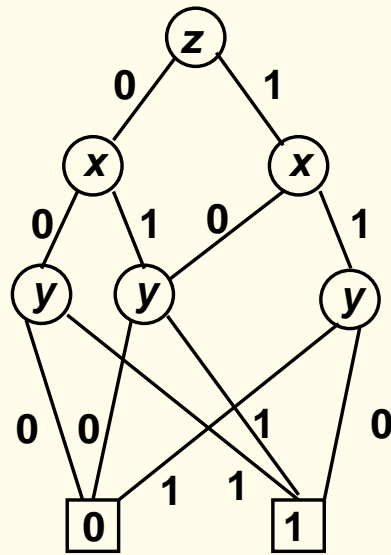
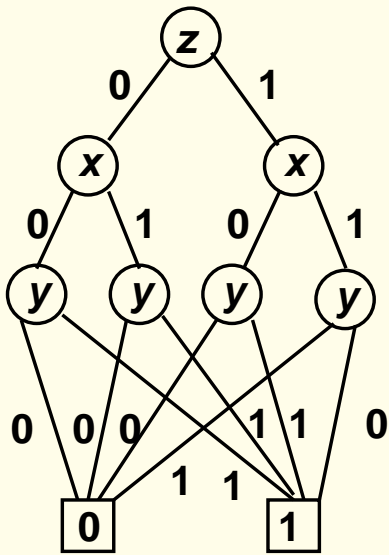
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



Operations with BDDs

$f \mapsto B_f$ (BDD associated with f)

$g \mapsto B_g$ (BDD associated with g)

BDD for $f \wedge g$: replace all 1-leaves in B_f with B_g

BDD for $f \vee g$: replace all 0-leaves in B_f with B_g

BDD for $\neg f$: replace all 1-leaves in B_f with 0-leaves and all 0-leaves with 1 leaves.

Binary Decision Diagrams

Binary decision diagram (BDD): finite directed acyclic graph with:

- a unique initial node
- terminal nodes marked with 0 or 1
- non-terminal nodes marked with propositional variables
- in each non-terminal node: two vertices (marked 0/1)

Reduced BDD: Optimizations 1-3 cannot be applied.

Binary Decision Diagrams

Binary decision diagram (BDD): finite directed acyclic graph with:

- a unique initial node
- terminal nodes marked with 0 or 1
- non-terminal nodes marked with propositional variables
- in each non-terminal node: two vertices (marked 0/1)

Reduced BDD: Optimizations 1-3 cannot be applied.

Problem: Variables may occur several times on a path.

Solution: Ordered BDDs.

Ordered BDDs

$[P_1, \dots, P_n]$ ordered list of variables (without repetitions)

Let B be a BDD with variables $\{P_1, \dots, P_n\}$

B has the order $[P_1, \dots, P_n]$

if for every path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_m$ in B ,

if - $i < j$,

- v_i is marked with P_{k_i}

- v_j is marked with P_{k_j}

then $k_i < k_j$.

A **ordered BDD** (Notation: OBDD) is a BDD which has an order, for a certain ordered list of variables.

Reduced OBDDs

Let $[P_1, \dots, P_n]$ be an order on variables.

The reduced OBDD, which represents a given function f is unique.

Theorem:

Let B_1, B_2 be two reduced OBDDs with the same variable ordering.

If B_1 and B_2 represent the same function, then B_1 and B_2 are equal.

OBDDs have a canonical form, namely the reduced OBDD.

Advantages of canonical representations

- Absence of redundant variables

If the value of f does not depend on the i -argument (P_i) then no reduced OBDD contains the variable P_i

- Equivalence test

$F_i \mapsto f_i \mapsto B_i$ (OBDDs with compatible variable ordering), $i = 1, 2$

Reduce B_i , $i = 1, 2$. $F_1 \equiv F_2$ iff. B_1 and B_2 identical.

Advantages of canonical representations

- Validity test

$F \mapsto f \mapsto B$ (OBDD)

F valid iff its reduced OBDD is $B_1 := \boxed{1}$

- Entailment test

$F \models G$ iff the reduced OBDD for $F \wedge \neg G$ is $B_0 := \boxed{0}$

- Satisfiability test

F satisfiable iff its reduced OBDD is not B_0 .

Operations with OBDDs

- Reduce

Apply reduction steps 1–3

- Apply

Boolean operations

- Restrict

Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

Compute OBDD for $\exists P_i F(P_1, \dots, P_n)$

Operations with OBDDs

- Reduce

Apply reduction steps 1–3

- Apply

Boolean operations

- Restrict

Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

Compute OBDD for $\exists P_i F(P_1, \dots, P_n)$

Reduce

remove redundancies

1. remove duplicate leaves
2. remove unnecessary tests
3. remove duplicate nodes

Reduce

The algorithm reduce traverses an OBDD B layer by layer in a bottom-up fashion, beginning with the terminal nodes.

In traversing B , it assigns an integer label $id(n)$ to each node n of B , in such a way that the subOBDDs with root nodes n and m denote the same boolean function iff, $id(n) = id(m)$.

Reduce

Terminal nodes:

Since reduce starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction 1).

Similarly, the 1-nodes all get the next label, say #1.

Reduce

Non-terminal nodes

Now let us inductively assume that reduce has already assigned integer labels to all nodes of a layer $> i$ (i.e. all terminal nodes and P_j -nodes with $j > i$).

We describe how nodes of layer i (i.e. P_i -nodes) are being handled.

$n \mapsto lo(n)$ node reached on branch labelled with 0

$hi(n)$ node reached on branch labelled with 1

Given an P_i -node n , there are three ways in which it may get its label:

- If $id(lo(n)) = id(hi(n))$, we set $id(n)$ to be that label (reduction 2)
- If there is another node m s.t. n and m have same variable P_i , and $id(lo(n)) = id(lo(m))$ and $id(hi(n)) = id(hi(m))$, then we set $id(n) := id(m)$ (reduction 3)
- Otherwise, we set $id(n)$ to the next unused integer label.

Operations with OBDDs

- Reduce

Apply reduction steps 1–3

- Apply

Boolean operations

- Restrict

Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

Compute OBDD for $\exists P_i F(P_1, \dots, P_n)$

Reminder: BDDs

$f \mapsto B_f$ (BDD associated with f)

$g \mapsto B_g$ (BDD associated with g)

BDD for $f \wedge g$: replace all 1-leaves in B_f with B_g

BDD for $f \vee g$: replace all 0-leaves in B_f with B_g

BDD for $\neg f$: replace all 1-leaves in B_f with 0-leaves and all 0-leaves with 1 leaves.

Reminder: BDDs

$f \mapsto B_f$ (BDD associated with f)

$g \mapsto B_g$ (BDD associated with g)

BDD for $f \wedge g$: replace all 1-leaves in B_f with B_g

BDD for $f \vee g$: replace all 0-leaves in B_f with B_g

BDD for $\neg f$: replace all 1-leaves in B_f with 0-leaves and all 0-leaves with 1 leaves.

If applied to OBDDs, the resulting BDD is not ordered!

Apply

Idea: Use the Shannon expansion for F .

$$F \equiv (\neg P \wedge F[0/P]) \vee (P \wedge F[1/P])$$

The function **apply** is based on the Shannon expansion for $F \text{ op } G$:

$$F \text{ op } G = (\neg P_i \wedge (F[0/P_i] \text{ op } G[0/P_i])) \vee (P_i \wedge (F[1/P_i] \text{ op } G[1/P_i])).$$

Apply

This is used as a control structure of apply which proceeds from the roots of B_F and B_G downwards to construct nodes of the OBDD $B_{F \text{ op } G}$.

Let r_f be the root node of B_F and r_g the root node of B_G .

1. If both r_f, r_g are terminal nodes with labels l_f and l_g , respectively (0 or 1), we compute the value $l_f \text{ op } l_g$ and let the resulting OBDD be B_0 if the value is 0 and B_1 otherwise.

Apply

This is used as a control structure of apply which proceeds from the roots of B_F and B_G downwards to construct nodes of the OBDD $B_{F \text{ op } G}$.

Let r_f be the root node of B_F and r_g the root node of B_G .

In the remaining cases, **at least one of the root nodes is a non-terminal.**

2. Suppose that **both root nodes are P_i -nodes.**

Then we create an P_i -node n with

- the edge labelled with 0 to $\text{apply}(\text{op}, lo(r_f), lo(r_g))$
- the edge labelled with 1 to $\text{apply}(\text{op}, hi(r_f), hi(r_g))$

Apply

This is used as a control structure of apply which proceeds from the roots of B_F and B_G downwards to construct nodes of the OBDD $B_{F \text{ op } G}$.

Let r_f be the root node of B_F and r_g the root node of B_G .

3. If r_f is a P_i -node, but r_g is a terminal node or a P_j -node with $j > i$, then we know that there is no P_i -node in B_G (because the two OBDDs have a compatible ordering of boolean variables).

Thus, G is independent of P_i ($G \equiv G[0/P_i] \equiv G[1/P_i]$).

Therefore, we create a P_i -node n with: - the 0-edge to $\text{apply}(\text{op}, \text{lo}(r_f), r_g)$ and
- the 1-edge to $\text{apply}(\text{op}, \text{hi}(r_f), r_g)$.

4. The case in which r_g is a non-terminal, but r_f is a terminal or a P_j -node with $j > i$, is handled symmetrically to case 3.

Apply

The result of this procedure might not be reduced; therefore apply finishes by calling the function reduce on the OBDD it constructed.

Restrict

Given an OBDD B_F representing a boolean formula F , we need an algorithm `restrict` such that:

– `restrict(0, P , B_F)` computes the reduced OBDD for $F[0/P]$ using the same variable ordering as B_F .

The algorithm works as follows.

For each node n labelled with P , incoming edges are redirected to $lo(n)$ and n is removed.

Then we call `reduce` on the resulting OBDD.

The call `restrict(1, P , B_F)` proceeds similarly, only we now redirect incoming edges to $hi(n)$.

Operations with OBDDs

- Reduce

Apply reduction steps 1–3

- Apply

Boolean operations

- Restrict

Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

Compute OBDD for $\exists P_i F(P_1, \dots, P_n)$

Exists

A boolean function can be thought of as putting a constraint on the values of its argument variables.

It is useful to be able to express the relaxation of the constraint on a subset of the variables concerned.

To allow this, we write $\exists P.F$ for the boolean function F with the constraint on P relaxed.

Formally, $\exists P.F$ is defined as $F[0/P] \vee F[1/P]$

that is, $\exists P.F$ is true if F could be made true by putting P to 0 or to 1.

Exists

Formally, $\exists P.F$ is defined as $F[0/P] \vee F[1/P]$

that is, $\exists P.F$ is true if F could be made true by putting P to 0 or to 1.

Therefore the **exists** algorithm can be implemented in terms of the algorithms **apply** and **restrict** as:

$$\text{exists}(P, F) := \text{apply}(\vee, \text{restrict}(0, P, B_F), \text{restrict}(1, P, B_F))$$

forall

Formally, $\forall P.F$ is defined as $F[0/P] \wedge F[1/P]$

that is, $\forall P.F$ is true if F could be made true both by putting P to 0 and by putting P to 1.

Therefore the **forall** algorithm can be implemented in terms of the algorithms **apply** and **restrict** as:

$$\text{forall}(P, F) := \text{apply}(\wedge, \text{restrict}(0, P, B_F), \text{restrict}(1, P, B_F))$$

Examples

Examples can be found on pages 378-379 of the book of the book Logic in Computer Science“by Huth and Ryan.

Limitations of Propositional Logic

- Fixed, finite number of objects
Cannot express: let G be group with arbitrary number of elements
- No functions or relations with arguments
Can express: finite function/relation table p_{ij}
Cannot express: properties of function/relation on all arguments,
e.g., $+$ is associative
- Static interpretation
Programs change value of their variables, e.g., via assignment, call,
etc.
Propositional formulas look at one single interpretation at a time

Beyond the Limitations of Propositional Logic

- First order logic
(+ functions)

- Temporal logic
(+ computations)

- Dynamic logic
(+ computations + functions)

Beyond the Limitations of Propositional Logic

- First order logic
(+ functions)
- Temporal logic
(+ computations)
- Dynamic logic
(+ computations + functions)

Part 2: First-Order Logic

Syntax:

- non-logical symbols (domain-specific)
⇒ terms, atomic formulas
- logical symbols (domain-independent)
⇒ Boolean combinations, quantifiers

Signature

A signature $\Sigma = (\Omega, \Pi)$, fixes an alphabet of non-logical symbols, where

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

If $n = 0$ then f is also called a **constant (symbol)**.

If $m = 0$ then p is also called a **propositional variable**.

Many-sorted Signature A many-sorted signature $\Sigma = (S, \Omega, \Pi)$, fixes an alphabet of non-logical symbols, where

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

where s_1, \dots, s_n, s_m, s are sorts.

Variables

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

We assume that

X

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

Many-sorted case:

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

Terms

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

$$\begin{array}{l} t, u, v ::= x, x \in X \quad \text{(variable)} \\ \quad \quad | f(t_1, \dots, t_n), f/n \in \Omega \quad \text{(functional term)} \end{array}$$

By $T_\Sigma(X)$ we denote the set of Σ -terms (over X).

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

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

Many-sorted case:

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

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

Atoms

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

$$A, B ::= p(t_1, \dots, t_m) \quad , p/m \in \Pi \\ \left[\quad \mid \quad (t \approx t') \quad \text{(equation)} \quad \right]$$

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

Many-sorted case:

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

Literals, Clauses

Literals

$L ::= A$ (positive literal)
| $\neg A$ (negative literal)

Clauses

$C, D ::= \perp$ (empty clause)
| $L_1 \vee \dots \vee L_k, k \geq 1$ (non-empty clause)

General First-Order Formulas

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

F, G, H	$::=$	\perp	(falsum)
		\top	(verum)
		A	(atomic formula)
		$\neg F$	(negation)
		$(F \wedge G)$	(conjunction)
		$(F \vee G)$	(disjunction)
		$(F \rightarrow G)$	(implication)
		$(F \leftrightarrow G)$	(equivalence)
		$\forall xF$	(universal quantification)
		$\exists xF$	(existential quantification)

Example: Peano Arithmetic

Signature:

$$\Sigma_{PA} = (\Omega_{PA}, \Pi_{PA})$$

$$\Omega_{PA} = \{0/0, +/2, */2, s/1\}$$

$$\Pi_{PA} = \{\leq /2, < /2\}$$

$$+, *, <, \leq \text{ infix; } * >_p + >_p < >_p \leq$$

Examples of formulas over this signature are:

$$\forall x, y (x \leq y \leftrightarrow \exists z (x + z \approx y))$$

$$\exists x \forall y (x + y \approx y)$$

$$\forall x, y (x * s(y) \approx x * y + x)$$

$$\forall x, y (s(x) \approx s(y) \rightarrow x \approx y)$$

$$\forall x \exists y (x < y \wedge \neg \exists z (x < z \wedge z < y))$$

Example: Specifying LISP lists

Signature:

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

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

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

Examples of formulae:

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

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

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

Many-sorted signatures

Example:

Signature

$$S = \{\text{array}, \text{index}, \text{element}\}$$

set of sorts

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

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

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

$$\Pi = \emptyset$$

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

Examples of formulae:

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

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