Intelligent Systems Programming

Lecture 6: **BDD Construction and Manipulation**

1. BDD construction
2. Boolean operations on BDDs
3. BDD-Based configuration

Today's Program

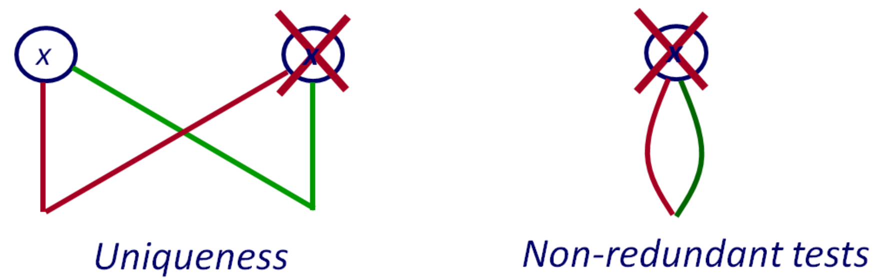
- [10:00-11:05]
  - Unique table
  - Build(*t*)
  - Apply(op,u_1,u_2)
- [11:15-12:00]
  - BDD-Based configuration
  - Henrik Hulgaard, Configit A/S

BDD Construction

BDD construction

Last week:

1. Make a Decision Tree of the Boolean expression
2. Keep reducing it until no further reductions are possible



This week:

- Reduce the decision tree to a BDD while building it

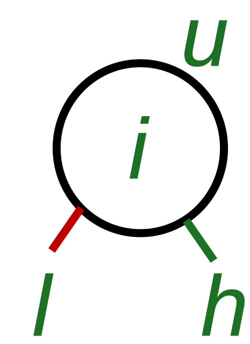
Reduce decision tree to BDD during construction

- Represent BDD by a table of unique nodes (*UT*)
- Build BDDs recursively,
  i.e. to add a new node *u:*
  1. Compute *high*(*u*) and *low*(*u*) and store them in *UT*
  2. Maintain BDD reductions when adding *u* to *UT*:
     a) Only extend *UT* with *u* if
        $high(u) \neq low(u)$ (non-redundancy test)
     b) Only extend *UT* with *u* if $u \notin UT$
        (uniqueness)

Unique Table Representation

Node Attributes

u	unique node identifier $\{\mathbf{0},\mathbf{1},2,3,\ldots\}$
i	variable index $\{1,2,\ldots,n,\boldsymbol{n}\mathbf{+1}\}$
l	node identifier of low
h	node identifier of high



Represent Unique Table by two tables *T* and *H*

$T : u \rightarrow (i,l,h)$

$H: (i,l,h) \rightarrow u$

H is the inverse of T:

$T(u) = (i,l,h) \Leftrightarrow H(i,l,h) = u$

Primitive Operations on T and H

$T : u \mapsto (i, l, h)$

$init(T)$	initialize T to contain only 0 and 1
$u \leftarrow add(T, i, l, h)$	allocate a new node u with attributes (i, l, h)
$var(u), low(u), high(u)$	lookup the attributes of u in T

$H : (i, l, h) \mapsto u$

$init(H)$	initialize H to be empty
$b \leftarrow member(H, i, l, h)$	check if (i, l, h) is in H
$u \leftarrow lookup(H, i, l, h)$	find $H(i, l, h)$
$insert(H, i, l, h, u)$	make (i, l, h) map to u in H

Unique Table Interface: MakeNode (MK)

$\textsc{Mk}[T, H](i, l, h)$

- $\textbf{if } l = h \textbf{ then return } l$
- $\textbf{else if } member(H, i, l, h) \textbf{ then}$
  - $\textbf{return } lookup(H, i, l, h)$
- $\textbf{else } u \leftarrow add(T, i, l, h)$
  - $insert(H, i, l, h, u)$
  - $\textbf{return } u$

Build

Idea: Construct the BDD recursively using the Shannon Expansion $t = x \rightarrow t[1/x], t[0/x]$

- Terminal cases

  Build(0) = 0

  Build(1) = 1

- Recursive case

  $\mathrm{Build}(t(x_i, x_{i+1}, \ldots, x_n)) = \mathrm{Mk}($ x_i $)$

  $\mathrm{Build}(t(0, x_{i+1}, \ldots, x_n))$ $\quad$ $\mathrm{Build}(t(1, x_{i+1}, \ldots, x_n))$

Build

$\text{BUILD}[T, H](t)$

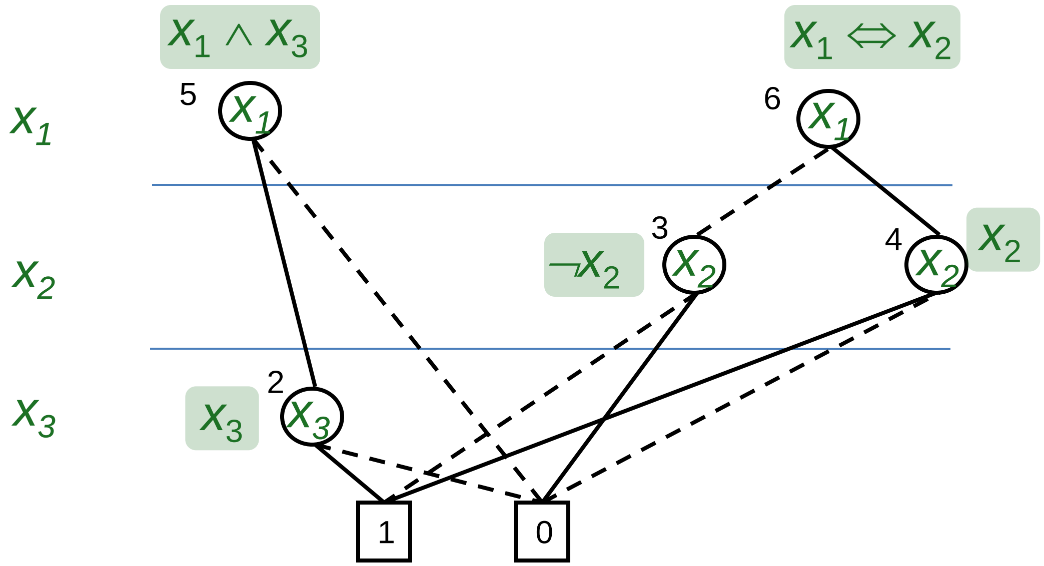
function $\text{BUILD'}(t, i) =$
    if $i > n$ **then**
        if t is false **then return** 0 **else return** 1
    else $v_0 \leftarrow \text{BUILD'}(t[0/x_i], i+1)$
        $v_1 \leftarrow \text{BUILD'}(t[1/x_i], i+1)$
        return $\text{MK}(i, v_0, v_1)$
end BUILD'

return $\text{BUILD'}(t, 1)$

BDD Manipulation

Multi-Rooted BDD

Unique Table contains many BDDs



Apply

- Apply(op,u_1,u_2): computes the BDD of

$$u_1 \; op \; u_2$$

where

op : any of the 16 Boolean operators

u_1, u_2: root nodes of BDDs

- Relies on the Shannon expansion properties:

$$(x \rightarrow t_1, t_0) \; op \; (x \rightarrow t'_1, t'_0) \;\; \equiv \;\; x \rightarrow (t_1 \; op \; t'_1), (t_0 \; op \; t'_0)$$

$$(x \rightarrow t_1, t_0) \; op \; t \;\; \equiv \;\; x \rightarrow (t_1 \; op \; t), (t_0 \; op \; t)$$

Apply with *op* = $\wedge$

- **Terminal case:** $u \in \{0,1\}$
  $u' \in \{0,1\}$

App($u \wedge u'$) $= u \wedge u'$

- **Recursive case:** $u = x_v \rightarrow u_1, u_0$
  $u' = x_w \rightarrow u'_1, u'_0$

App($u \wedge u'$) =

Mk(x_v, App($u_0 \wedge u'_0$), App($u_1 \wedge u'_1$)) if $v = w$

Mk(x_v, App($u_0 \wedge u'$), App($u_1 \wedge u'$)) if $v < w$

Mk(x_w, App($u \wedge u'_0$), App($u \wedge u'_1$)) if $w < v$

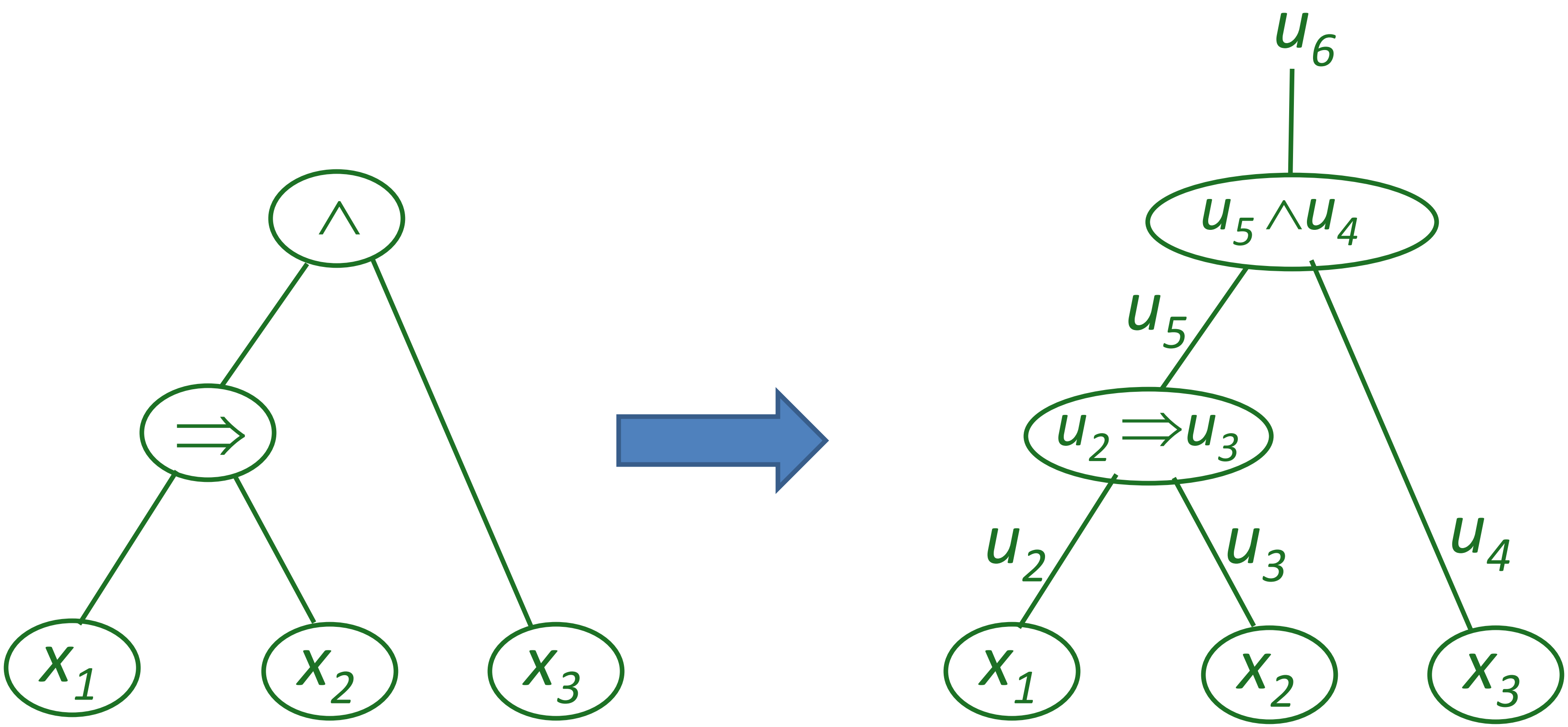
$\textsc{Apply}[T,H](op, u_1, u_2)$

```
init(G)

function APP(u1, u2) =
   if G(u1, u2) ≠ empty then return G(u1, u2)
   else if u1 ∈ {0,1} and u2 ∈ {0,1} then u ← op(u1, u2)
   else if var(u1) = var(u2) then
         u ← MK(var(u1), APP(low(u1), low(u2)), APP(high(u1), high(u2)))
   else if var(u1) < var(u2) then
         u ← MK(var(u1), APP(low(u1), u2), APP(high(u1), u2))
   else (* var(u1) > var(u2) *)
         u ← MK(var(u2), APP(u1, low(u2)), APP(u1, high(u2)))
   G(u1, u2) ← u
   return u
end APP

return APP(u1, u2)
```

Construct BDDs from expression tree



Properties of Apply

- Improvements?
  - Early termination. E.g., no reason to keep recursing if the left side in a conjunction is 0
- Complexity : $O(|u_1||u_2|)$, due to dynamic programming
- So a BDD of any formula can be computed in poly time?

BDDs

- Compact
- Equality check easy
- Easy to evaluate the truth-value of an assignment
- Boolean operations efficient
- SAT check efficient
- Tautology check efficient
- Easy to implement