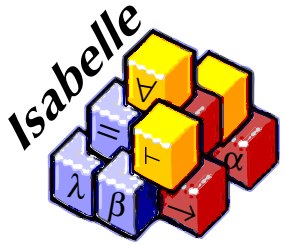
TYPES '04

Verification of BDD Algorithms

Veronika Ortner
Norbert Schirmer

Institut für Informatik, TU München

16th December 2004



Content

Content

1. Subject and Goals

Content

1. Subject and Goals

2. The Verification Environment

Content

1. Subject and Goals
2. The Verification Environment
3. Verified BDD Algorithms

Content

1. Subject and Goals
2. The Verification Environment
3. Verified BDD Algorithms
4. Results

Subject and Goals

- Goals:
  - Verification of pointer-based data structures
  - Formal proof of the normalization algorithm on BDDs

Subject and Goals

- Goals:
  - Verification of pointer-based data structures
  - Formal proof of the normalization algorithm on BDDs

- Approach:
  - Formalization of BDDs and their properties in Isabelle/Isar
  - Implementation of BDD algorithms in a C-like language
  - Specification and verification using our verification environment for Hoare Logic

Basics of the Verification Environment

- Definition of a **record** for program variables and the heap

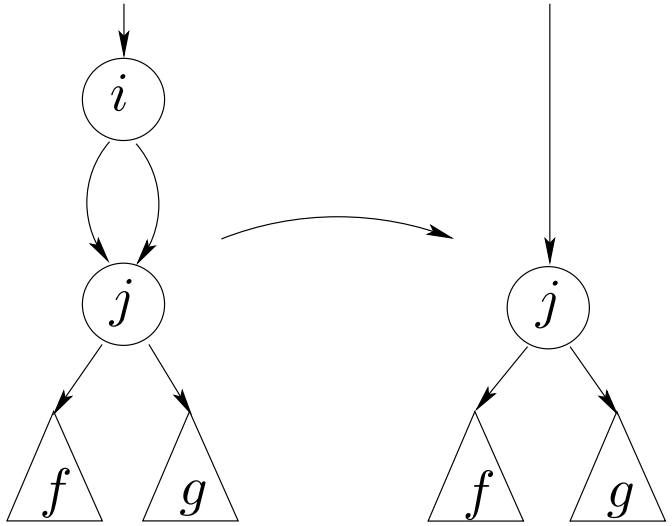
Basics of the Verification Environment

- Definition of a **record** for program variables and the heap
- Hoare-annotated program $\xrightarrow{vcg}$ proof obligation

  This proof obligation is the basis for our verification work.

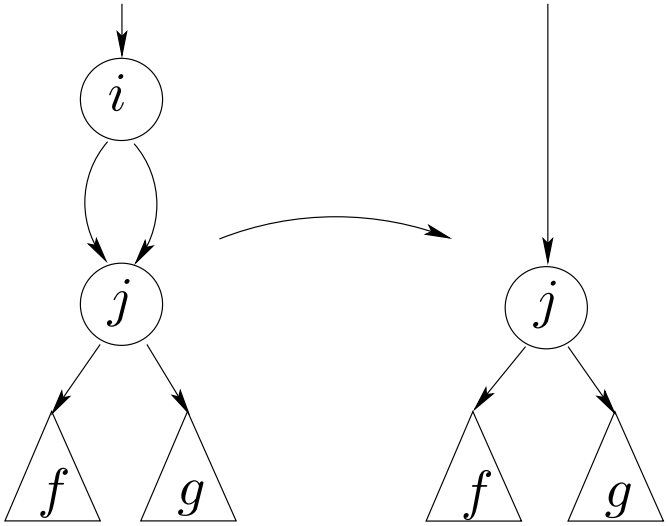
Properties of BDDs

reduced:

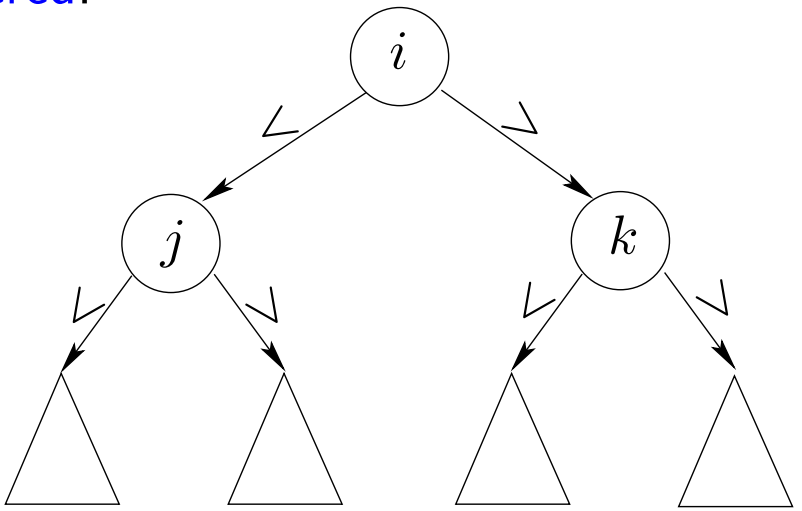


Properties of BDDs

reduced:

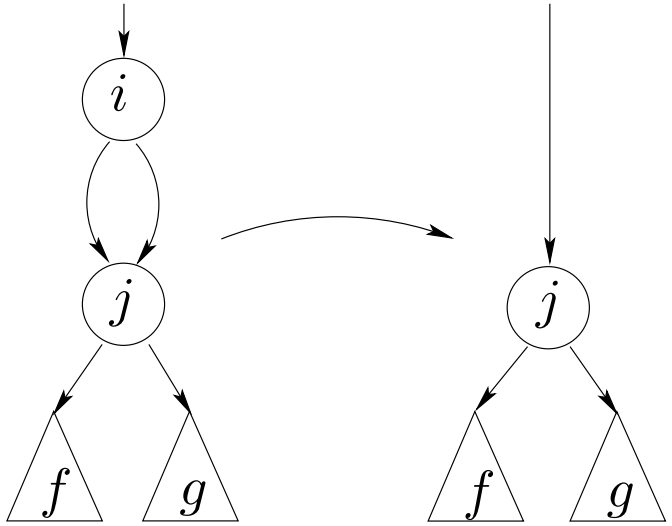


ordered:

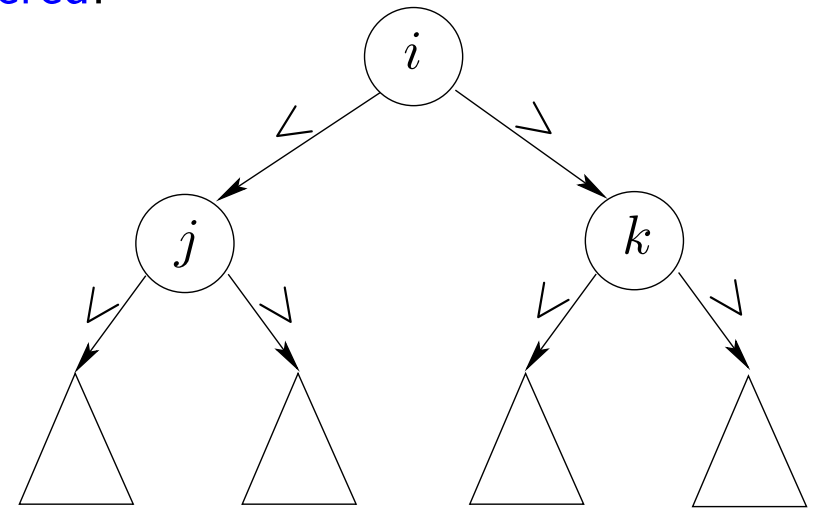


Properties of BDDs

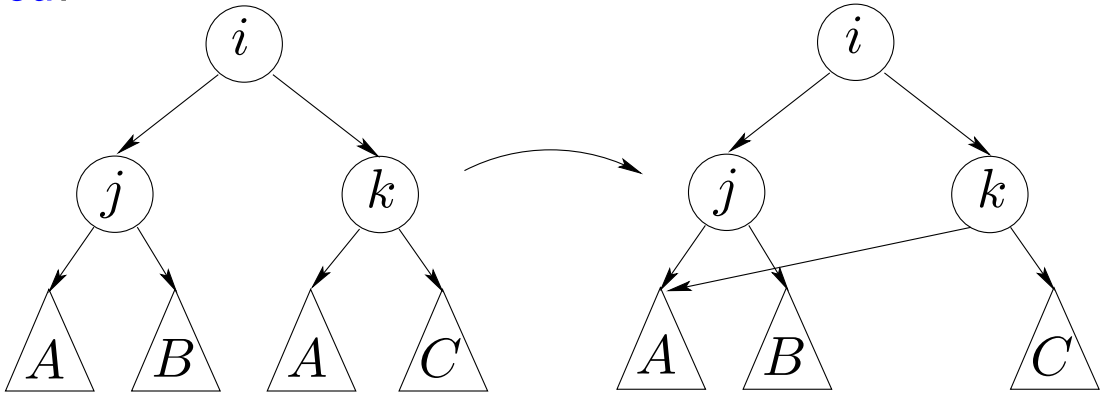
reduced:



ordered:

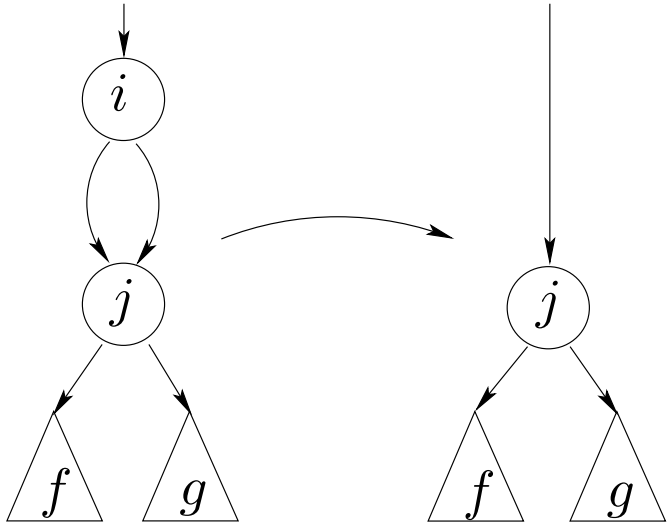


shared:

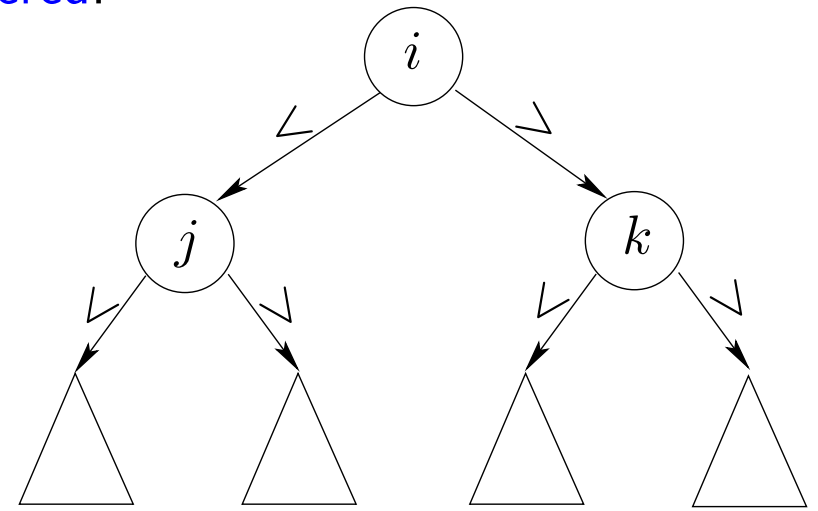


Properties of BDDs

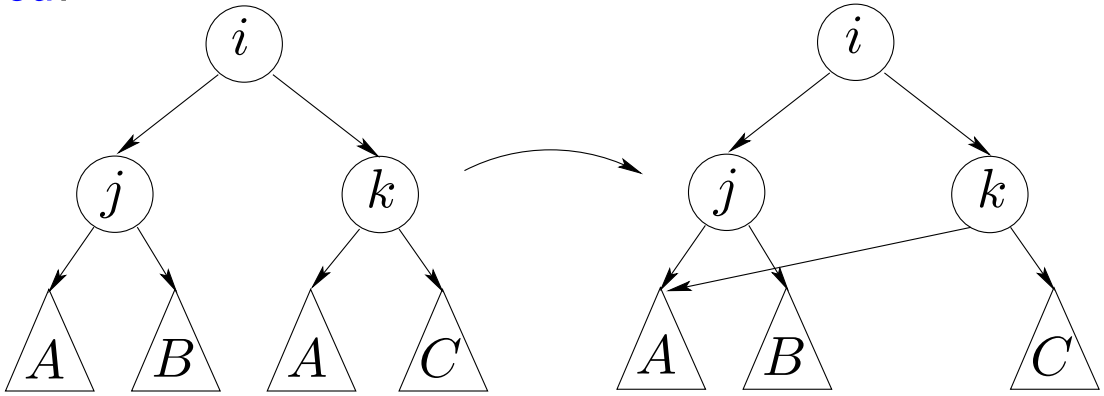
reduced:



ordered:



shared:

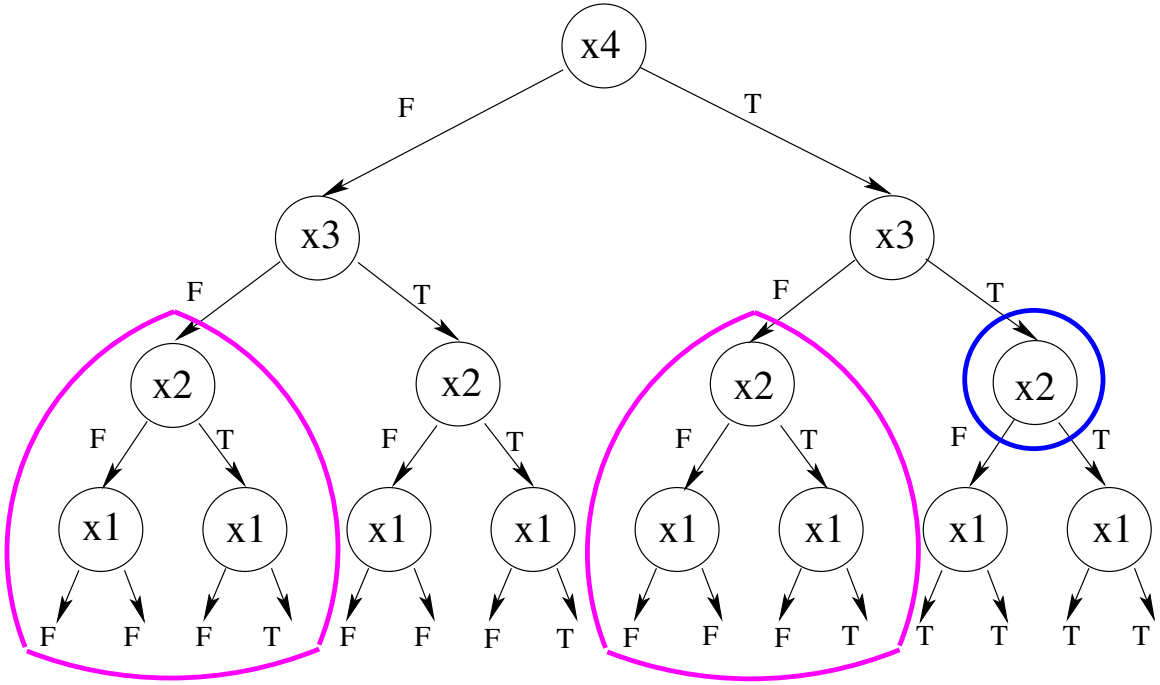


Result:

- Unique representation for each function
- No redundant parts

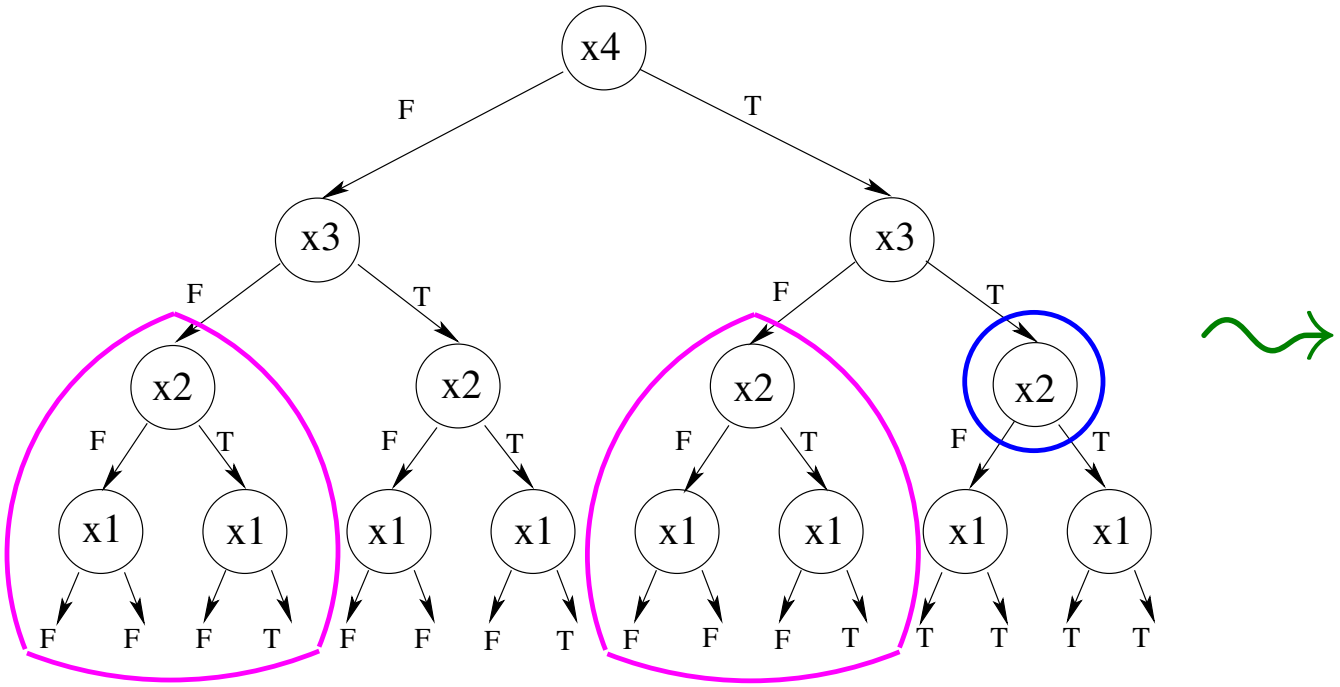
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



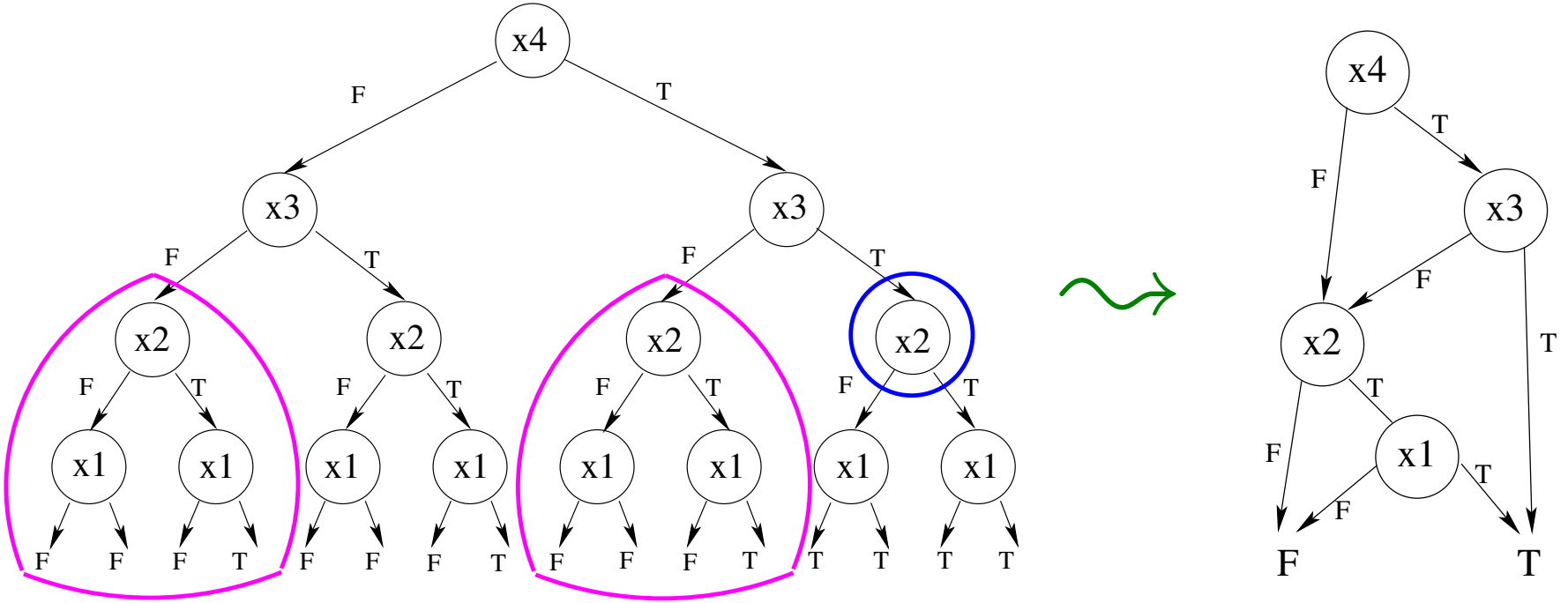
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



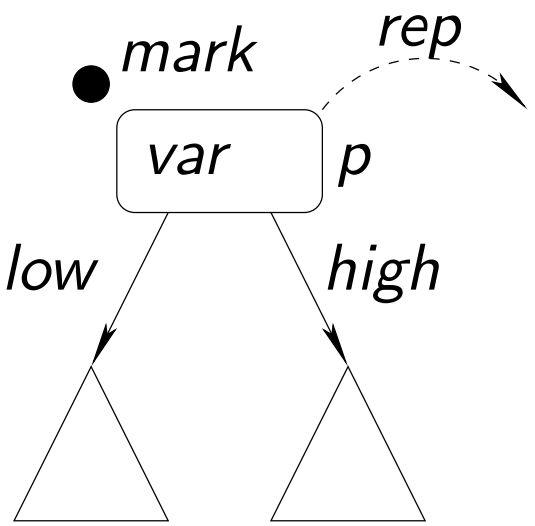
Example

$$f(x1, x2, x3, x4) := x1 \wedge x2 \vee x3 \wedge x4$$



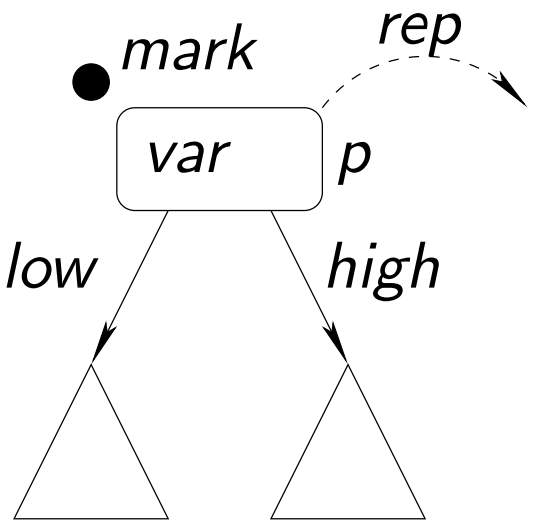
Representation of BDDs in the heap

```
struct {
   nat var;
   node* low;
   node* high;
   node* rep;
   bool mark;
} node;
```



Representation of BDDs in the heap

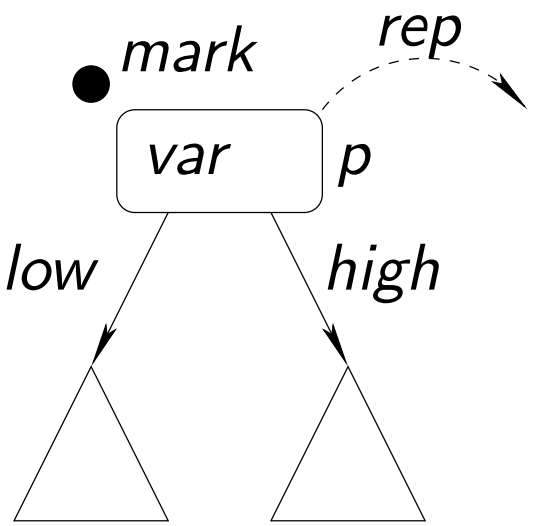
```
struct {
   nat var;
   node* low;
   node* high;
   node* rep;
   bool mark;
} node;
```



record *bdd_data* =
  var :: *ref* ⇒ *nat*
  low :: *ref* ⇒ *ref*
  high :: *ref* ⇒ *ref*
  ref :: *ref* ⇒ *ref*
  mark :: *ref* ⇒ *bool*
  ⋮

Representation of BDDs in the heap

```
struct {
   nat var;
   node* low;
   node* high;
   node* rep;
   bool mark;
} node;
```



record $bdd\_data =$
$var :: ref \Rightarrow nat$
$low :: ref \Rightarrow ref$
$high :: ref \Rightarrow ref$
$ref :: ref \Rightarrow ref$
$mark :: ref \Rightarrow bool$
$\vdots$

bdd_data shows split heap approach
Advantage:
Specification of properties on relevant heap components instead of on whole heap

BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:
  datatype $dag = Tip \mid Node\ dag\ ref\ dag$

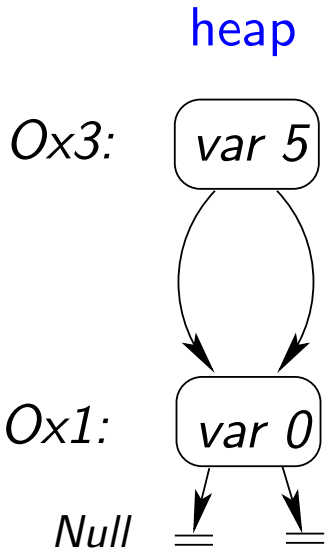
- Decision tree (e.g. for evaluation purposes):
  datatype $bdt = Zero \mid One \mid Bdt\_Node\ bdt\ nat\ bdt$

BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:
  datatype $dag = Tip \mid Node\ dag\ ref\ dag$

- Decision tree (e.g. for evaluation purposes):
  datatype $bdt = Zero \mid One \mid Bdt\_Node\ bdt\ nat\ bdt$

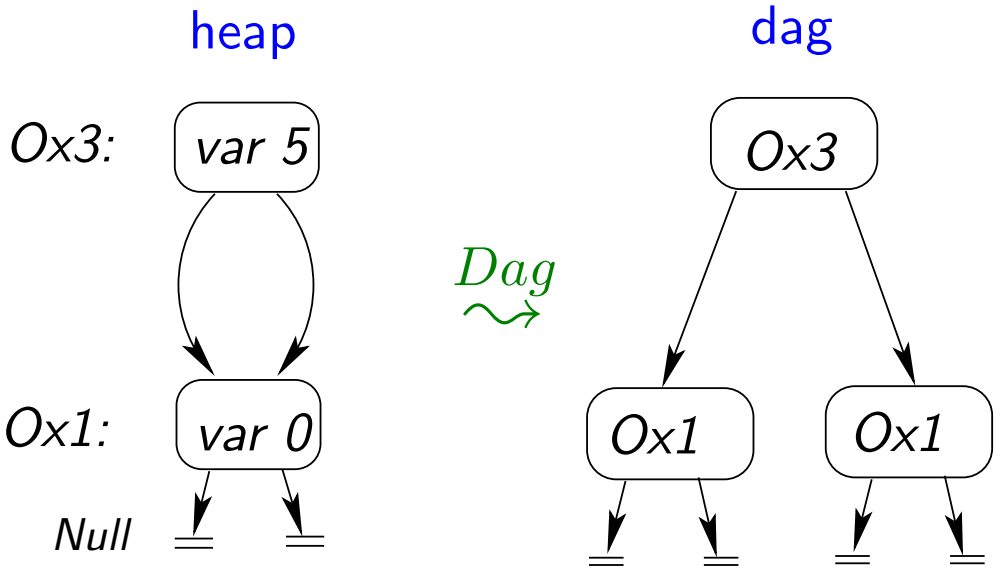


BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:
  datatype $dag = Tip \mid Node\ dag\ ref\ dag$

- Decision tree (e.g. for evaluation purposes):
  datatype $bdt = Zero \mid One \mid Bdt\_Node\ bdt\ nat\ bdt$

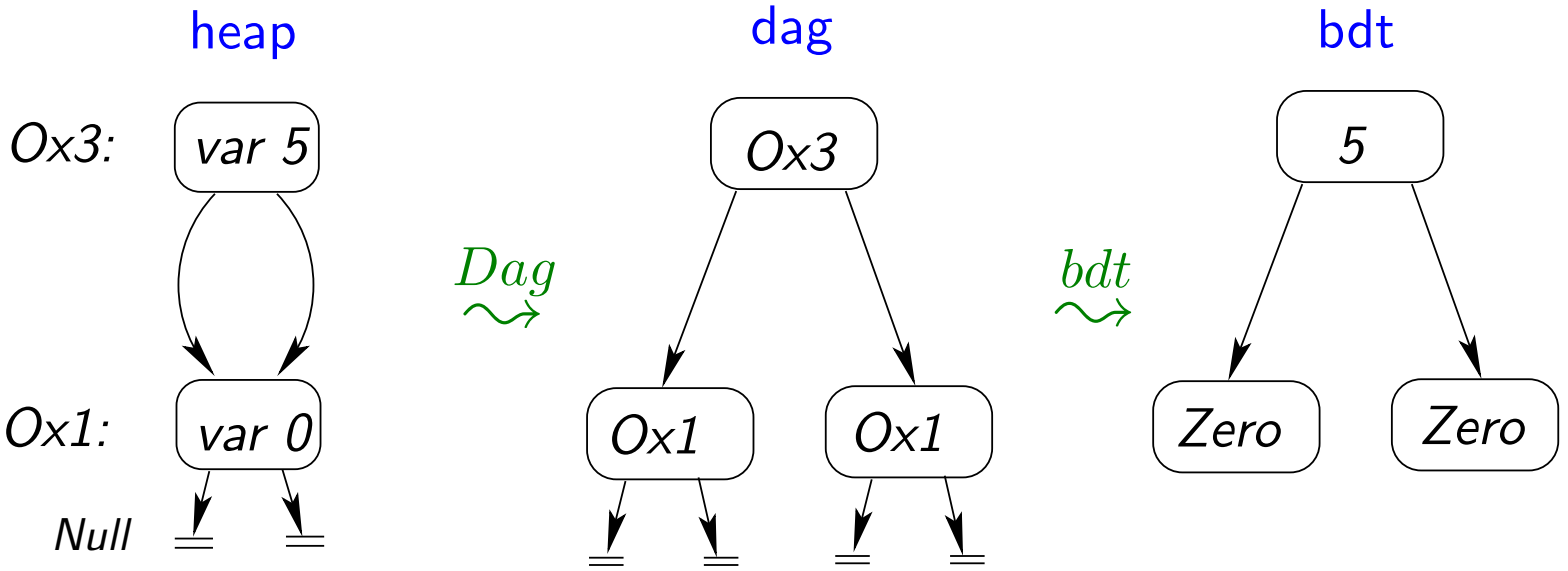


BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:
  datatype $dag = Tip \mid Node\ dag\ ref\ dag$

- Decision tree (e.g. for evaluation purposes):
  datatype $bdt = Zero \mid One \mid Bdt\_Node\ bdt\ nat\ bdt$

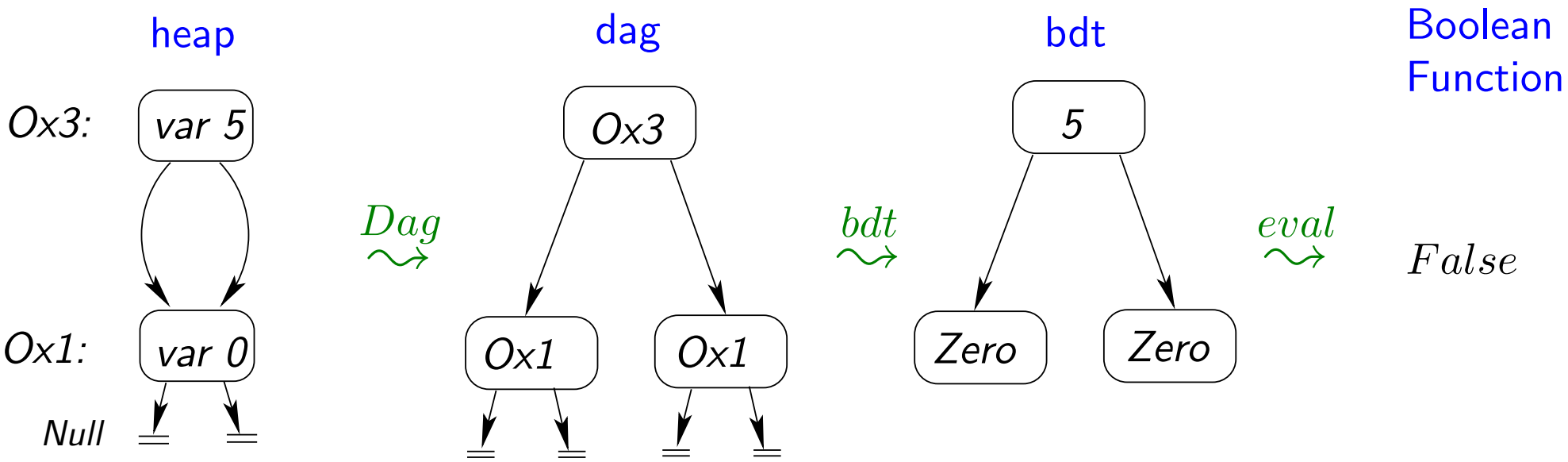


BDD Formalization - Illustration

2 levels of specification:

- Graph structure in order to represent *sharing*:
  datatype $dag = Tip \mid Node\ dag\ ref\ dag$

- Decision tree (e.g. for evaluation purposes):
  datatype $bdt = Zero \mid One \mid Bdt\_Node\ bdt\ nat\ bdt$



Transition Functions

- Construction of a dag in the heap:
  consts $Dag :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow (ref \Rightarrow ref) \Rightarrow dag \Rightarrow bool$
  primrec
  $Dag\ p\ low\ high\ Tip = (p = Null)$
  $Dag\ p\ low\ high\ (Node\ ldag\ a\ rdag) = (p = a \wedge p \neq Null \wedge$
  $Dag\ (low\ p)\ low\ high\ ldag \wedge Dag\ (high\ p)\ low\ high\ rdag)$

Transition Functions

- Construction of a dag in the heap:
  consts $Dag :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow (ref \Rightarrow ref) \Rightarrow dag \Rightarrow bool$
  primrec
  $Dag\ p\ low\ high\ Tip = (p = Null)$
  $Dag\ p\ low\ high\ (Node\ ldag\ a\ rdag) = (p = a \wedge p \neq Null \wedge$
  $Dag\ (low\ p)\ low\ high\ ldag \wedge Dag\ (high\ p)\ low\ high\ rdag)$

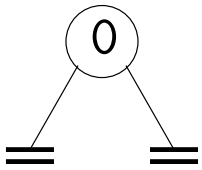
- Construction of a bdt out of a dag:
  consts $bdt :: dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bdt\ option$

Problems with the bdt Conversion

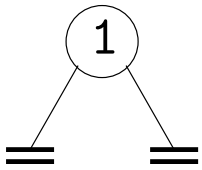
Reasons, why the conversion could go wrong:

- $bdt\ Tip\ var = ?$
- bdt-endnodes ($Zero/One$) must be represented by an inner node in the dag-representation:

  $Zero$ node:

  One node:

- No inner bdt-node may contain variable values 0 and 1
- Balanced structure of bdt-nodes

Normalize - Auxiliary Functions for the Specification

- eval:
  consts $eval :: bdt \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

Normalize - Auxiliary Functions for the Specification

- eval:
  consts $eval :: bdt \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

- cong_eval ($\sim$):
  $bdt_1 \sim bdt_2 \equiv (eval\ bdt_1 = eval\ bdt_2)$

Normalize - Auxiliary Functions for the Specification

- eval:
  consts $eval :: bdt \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

- cong_eval ($\sim$):
  $bdt_1 \sim bdt_2 \equiv (eval\ bdt_1 = eval\ bdt_2)$

- isomorphic_dags_eq:
  $isomorphic\_dags\_eq\ dag_1\ dag_2\ var \equiv$
  $\forall bdt_1\ bdt_2.\ (bdt\ dag_1\ var = Some\ bdt_1 \wedge bdt\ dag_2\ var = Some\ bdt_2 \wedge$
  $(bdt_1 = bdt_2)) \longrightarrow dag_1 = dag_2$

Normalize - Auxiliary Functions for the Specification

- eval:
  consts $eval :: bdt \Rightarrow (nat \Rightarrow bool) \Rightarrow bool$

- cong_eval ($\sim$):
  $bdt_1 \sim bdt_2 \equiv (eval\ bdt_1 = eval\ bdt_2)$

- isomorphic_dags_eq:
  $isomorphic\_dags\_eq\ dag_1\ dag_2\ var \equiv$
  $\forall bdt_1\ bdt_2.\ (bdt\ dag_1\ var = Some\ bdt_1 \wedge bdt\ dag_2\ var = Some\ bdt_2 \wedge$
  $(bdt_1 = bdt_2)) \longrightarrow dag_1 = dag_2$

- shared:
  $shared\ dag\ var \equiv$
  $\forall dag_1\ dag_2.\ (dag_1 \leq dag \wedge dag_2 \leq dag) \longrightarrow isomorphic\_dags\_eq\ dag_1\ dag_2\ var$

Normalize - Specification

```
‘levellist := replicate (‘p→‘var + 1) [];
‘levellist := CALL Levellist(‘p, ¬‘p→‘mark, ‘levellist);
‘n := 0;
WHILE ‘n < length ‘levellist
DO CALL ShareReduceRepList(‘levellist !  ‘n); ‘n := ‘n + 1 OD;
‘p := CALL Repoint(‘p)
```

Normalize - Specification

$\forall\ \sigma\ predag\ prebdt.\ \Gamma \vdash$
$\{\!|\sigma.\ Dag\ `p\ `low\ `high\ predag \wedge ordered\ predag\ `var \wedge$
$\quad bdt\ predag\ `var = Some\ prebdt\ |\!\}$

```
‘levellist := replicate (‘p→‘var + 1) [];
‘levellist := CALL Levellist(‘p, ¬‘p→‘mark, ‘levellist);
‘n := 0;
WHILE ‘n < length ‘levellist
DO CALL ShareReduceRepList(‘levellist !  ‘n); ‘n := ‘n + 1 OD;
‘p := CALL Repoint(‘p)
```

Normalize - Specification

$\forall\ \sigma\ predag\ prebdt.\ \Gamma \vdash$
$\{\!|\sigma.\ Dag\ `p\ `low\ `high\ predag \wedge ordered\ predag\ `var \wedge$
$\quad bdt\ predag\ `var = Some\ prebdt\ |\!\}$

```
‘levellist := replicate (‘p→‘var + 1) [];
‘levellist := CALL Levellist(‘p, ¬‘p→‘mark, ‘levellist);
‘n := 0;
WHILE ‘n < length ‘levellist
DO CALL ShareReduceRepList(‘levellist !  ‘n); ‘n := ‘n + 1 OD;
‘p := CALL Repoint(‘p)
```

$\{\!|\ (\exists\ postdag.\ Dag\ `p\ `low\ `high\ postdag \wedge reduced\ postdag \wedge shared\ postdag\ ^{\sigma}var \wedge$
$\quad ordered\ postdag\ ^{\sigma}var \wedge set\_of\ postdag \subseteq set\_of\ predag \wedge$
$\quad (\exists\ postbdt.\ bdt\ postdag\ ^{\sigma}var = Some\ postbdt \wedge prebdt \sim postbdt))|\!\}$

Results

Results

- Complete formalization of the BDD concept
- First machine-checked proof of the normalization algorithms
- Verification of C-like pointer programs possible using the verification environment for Hoare Logic in Isabelle

Thank you for your attention!