Correctness of Time-forward Processing Algorithms

Steffan Sølvsten, Simon Wimmer

LogSem Seminar, 2nd of December 2024

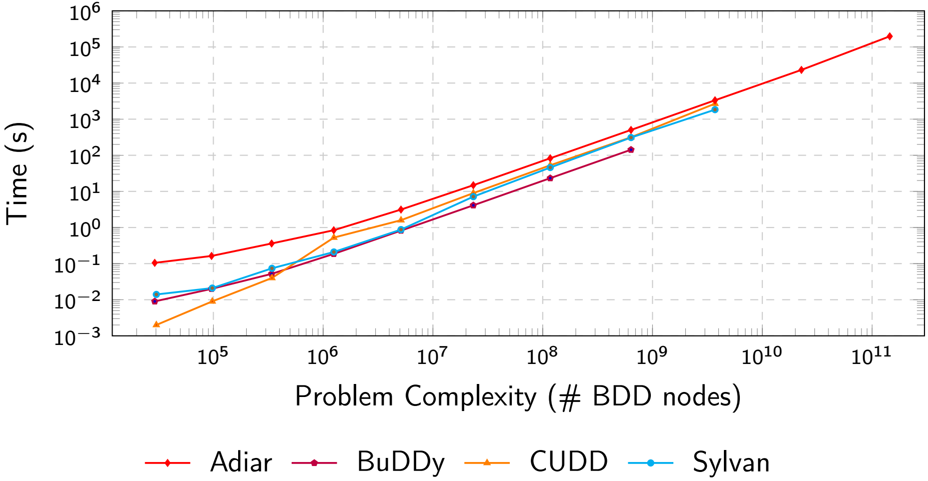
AARHUS UNIVERSITY

Adiar

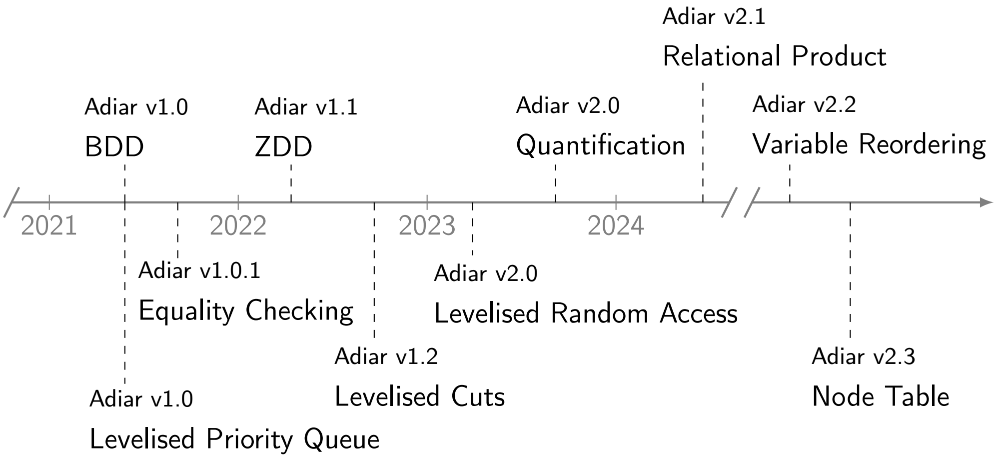
I/O-efficient Decision Diagrams

---

github.com/logsem/adiar



Running Time to solve *N-Queens* problems.


Adiar v2.1
Relational Product
Adiar v1.0
BDD
Adiar v1.1
ZDD
Adiar v2.0
Quantification
Adiar v2.2
Variable Reordering
2021
2022
2023
2024
Adiar v1.0.1
Equality Checking
Adiar v2.0
Levelised Random Access
Adiar v1.2
Levelised Cuts
Adiar v2.3
Node Table
Adiar v1.0
Levelised Priority Queue

Written in

C++

```
//////////////////////////////////////////////////////////////////
/// \brief Negates the content of ‘p‘ if it is a terminal and the
///        ‘negate‘ flag is set to true.
//////////////////////////////////////////////////////////////////

inline ptr_uint64
cnot(const ptr_uint64& p, const bool negate)
{
  const ptr_uint64::raw_type shifted_negate =
    ((ptr_uint64::raw_type) negate) << ptr_uint64::data_shift;
  return p.is_terminal() ? p._raw ^ shifted_negate : p._raw;
}
```

Correctness guaranteed by

$\sim$3500 Unit Tests

$\sim$400 Integration Tests

Cache and I/O Efficient Functional Algorithms

Guy E. Blelloch Robert Harper

Carnegie Mellon University

Abstract

In this paper we present a cost model for analyzing the memory efficiency of algorithms expressed in a simple functional language. We show how some algorithms written in standard forms using just lists and trees (no arrays) and requiring no explicit memory layout or memory management are efficient in the model. We then describe an implementation of the language and show provable bounds for mapping the cost in our model to the cost in the ideal- cache model. These bound imply that purely functional programs based on lists and trees with no special attention to any details of memory layout can be as asymptotically as efficient as the carefully designed imperative I/O efficient algorithms. For example we describe an $\mathcal{O}(N/B \log_{M/B} N/B)$ cost sorting algorithm, which is optimal in the ideal cache and I/O models.

Contents

Motivation

Correctness of Time-forward Processing

Encoding Binary Decision Diagrams

bbd_eval

bbd_not

bbd_satcount

Appendix

$\lfloor a \rfloor_n$: Bounded Domain

$\#_n$: Number of Assignments

Contents

Motivation

Correctness of Time-forward Processing

- Encoding Binary Decision Diagrams
- bbd_eval
- bbd_not
- bbd_satcount

Appendix

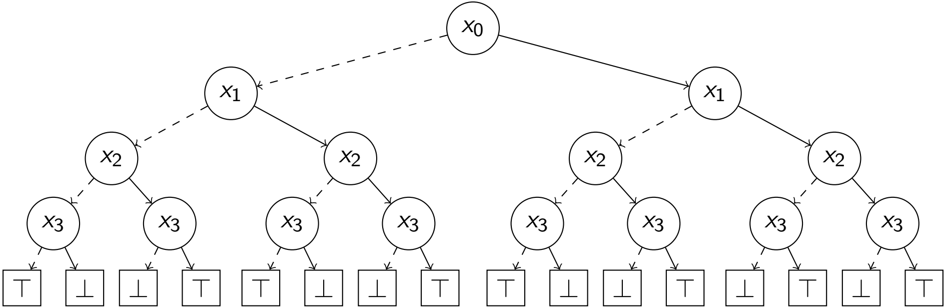
- $\lfloor a \rfloor_n$: Bounded Domain
- $\#_n$: Number of Assignments

Semantics of BDDs

$$f(x_0, x_1, x_2, x_3) \equiv (x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$$

[1] Julius Michaelis, Maximilian Haslbeck, Peter Lammich, and Lars Hupel. "Algorithms for Reduced Ordered Binary Decision Diagrams". In: Archive of Formal Proofs (2016)

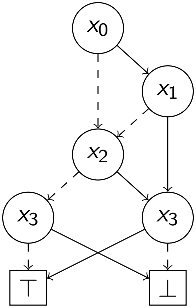
Semantics of BDDs



Binary Decision Tree

[1] Julius Michaelis, Maximilian Haslbeck, Peter Lammich, and Lars Hupel. "Algorithms for Reduced Ordered Binary Decision Diagrams". In: Archive of Formal Proofs (2016)

Semantics of BDDs



Binary Decision Diagram

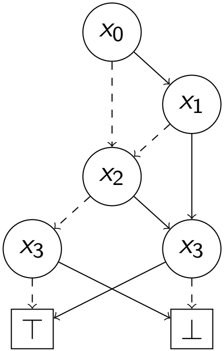
[1] Julius Michaelis, Maximilian Haslbeck, Peter Lammich, and Lars Hupel. "Algorithms for Reduced Ordered Binary Decision Diagrams". In: Archive of Formal Proofs (2016)

Data Types: Unique Identifiers and Pointers

```
Uid = (level: ℕ, id: ℕ)

operator (a: Uid) < (b: Uid) =
    a.level < b.level ∨ (a.level = b.level ∧ a.id < b.id)

Ptr = Leaf (val : 𝔹)
    | Node (uid : Uid)

operator (a: Ptr) < (b: Ptr) =
    lift Uid::< s.t. Ptr::Node < Ptr::Leaf
```

Lemma
Uid::< and Ptr::< are total orders.

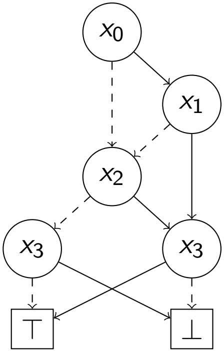
Data Types: Nodes and BDDs



$(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

```
Node = (i: Uid, t: Ptr, e: Ptr)
Bdd  = Leaf  (val : 𝔹)
     | Nodes (ns : List[Node])

Example = Bdd::Nodes([
  Node(Uid(0,0), Ptr(1,0), Ptr(2,0));
  Node(Uid(1,0), Ptr(3,1), Ptr(2,0));
  Node(Uid(2,0), Ptr(3,1), Ptr(3,0));
  Node(Uid(3,0), Ptr(⊥), Ptr(⊤));
  Node(Uid(3,1), Ptr(⊤), Ptr(⊥));
])
```

Data Types: Nodes and BDDs



$(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

```
Node = (i: Uid, t: Ptr, e: Ptr)
Bdd  = Leaf  (val : 𝔹)
     | Nodes (ns : List[Node])
```

Definition
A `Bdd` is *well formed*, if `ns` : `List[Node]` satisfies:

1. It is *non-empty*.
2. It is *closed*, i.e. every node referred to exists.
3. For each node, the level is strictly increasing.
4. It is *sorted* w.r.t. `Node`::`Uid`.

Contents

Motivation

Correctness of Time-forward Processing

- Encoding Binary Decision Diagrams
- bbd_eval
- bbd_not
- bbd_satcount

Appendix

- $\lfloor a \rfloor_n$: Bounded Domain
- $\#_n$: Number of Assignments

bdd_eval f x

```
bdd_eval (f: Bdd) (x: ℕ → 𝔹)
  let bdd_eval' (ns: List[Node]) (tgt: Uid) (x: ℕ → 𝔹) =
    match ns with
    | (i t e)::ns' =>
      if i < tgt
      then bdd_eval' ns' tgt x
      else bdd_eval' ns' (if x(a) then t else e) x
  in match f with
     | Leaf  v      => v
     | Nodes r::ns => bdd_eval' r::ns r x
```

bdd_eval f x

Define the function `bdt_of_bdd` : `Bdd` $\rightarrow$ `Bdt` to converts a Binary Decision Diagram into a Binary Decision Tree. Here, skip over "irrelevant" nodes to convert subtrees.

bdd_eval f x

Define the function `bdt_of_bdd` : `Bdd` $\to$ `Bdt` to converts a Binary Decision Diagram into a Binary Decision Tree. Here, skip over "irrelevant" nodes to convert subtrees.

Theorem
If f is well formed, then $\forall x$: `bdd_eval f x` $\iff$ `bdt_eval (bdt_of_bdd f) x`.

Proof.
Case `Leaf` $\mathbb{B}$: Trivial

Case `Nodes ns`:

Induction on `ns`.

Discard bad cases due to the BDD being *closed* and *sorted*. $\square$

Contents

Motivation

Correctness of Time-forward Processing

- Encoding Binary Decision Diagrams
- bbd_eval
- bbd_not
- bbd_satcount

Appendix

- $\lfloor a \rfloor_n$: Bounded Domain
- $\#_n$: Number of Assignments

bdd_not f

```
operator ptr (p: Ptr) = match p with
                      | Leaf v => !v
                      | Node u => u

bdd_not (f: Bdd) = match f with
                 | Leaf v   => !v
                 | Nodes ns => map (i t e) => (i !t !e) ns
```

bdd_not f

```
operator ptr (p: Ptr) = match p with
                      | Leaf v => !v
                      | Node u => u

bdd_not (f: Bdd) = match f with
                 | Leaf v   => !v
                 | Nodes ns => map (i t e) => (i !t !e) ns
```

Theorem
If f is well formed, then $\forall x$: $\neg$ *(bdd_eval f x)* $\iff$ *bff_eval (bdd_not f) x.*

Proof.
Case `Leaf` $\mathbb{B}$: Trivial

Case `Nodes ns`: Induction on `ns`. $\square$

Contents

Motivation

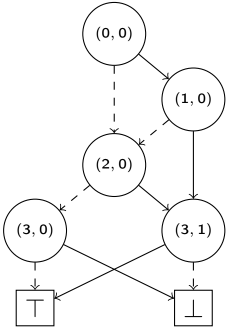
Correctness of Time-forward Processing

- Encoding Binary Decision Diagrams
- bbd_eval
- bbd_not
- bbd_satcount

Appendix

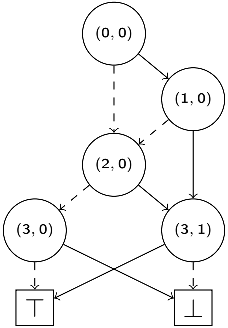
- $\lfloor a \rfloor_n$: Bounded Domain
- $\#_n$: Number of Assignments

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

bdd_pathcount f



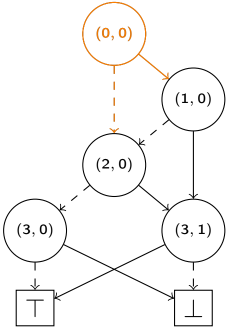
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

[

]

bdd_pathcount f



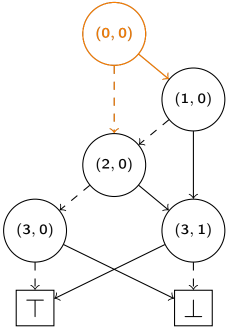
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

[

]

bdd_pathcount f

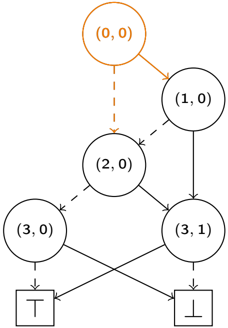


(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

$$[\quad ((0,0) \xrightarrow{\top} (1,0),\quad 1)\quad ,$$
$$((0,0) \xrightarrow{\bot} (2,0),\quad 1)\quad ,$$
$$]$$

bdd_pathcount f



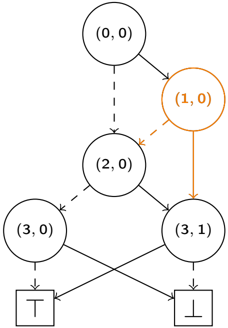
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
$(1, 0)$	0	0

Priority Queue: Q_{count}:

$$[\quad ((0,0) \xrightarrow{\top} (1,0), \quad 1) \quad , \\ ((0,0) \xrightarrow{\bot} (2,0), \quad 1) \quad , \\ ]$$

bdd_pathcount f



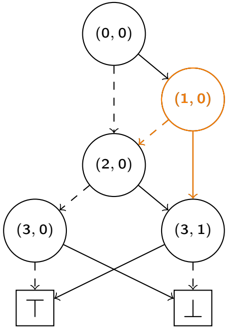
(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

Seek	Sum	Result
$(1, 0)$	0	0

Priority Queue: Q_{count}:

$$[\quad ((0,0) \xrightarrow{\top} (1,0), \quad 1) \quad , \quad ((0,0) \xrightarrow{\bot} (2,0), \quad 1) \quad , \quad]$$

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
$(1, 0)$	1	0

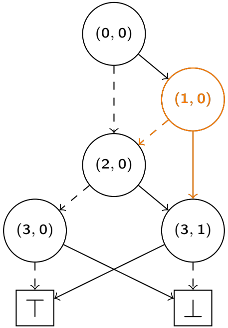
Priority Queue: Q_{count}:

[

$((0,0) \xrightarrow{\perp} (2,0), \quad 1)$,

]

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

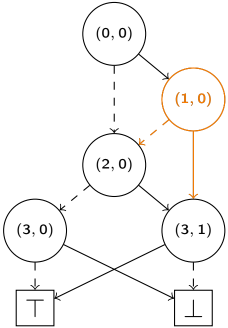
Seek	Sum	Result
$(1, 0)$	1	0

Priority Queue: Q_{count}:

[
$((0,0) \xrightarrow{\perp} (2,0),\quad 1)$,
$((1,0) \xrightarrow{\perp} (2,0),\quad 1)$,

$((1,0) \xrightarrow{\top} (3,1),\quad 1)$,
]

bdd_pathcount f



(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

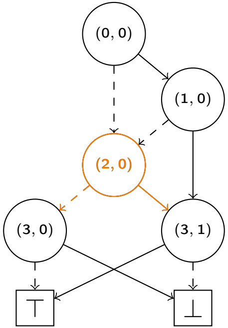
Seek	Sum	Result
$(2, 0)$	0	0

Priority Queue: Q_{count}:

[
$((0,0) \xrightarrow{\perp} (2,0),\quad 1)$,
$((1,0) \xrightarrow{\perp} (2,0),\quad 1)$,

$((1,0) \xrightarrow{\top} (3,1),\quad 1)$,
]

bdd_pathcount f



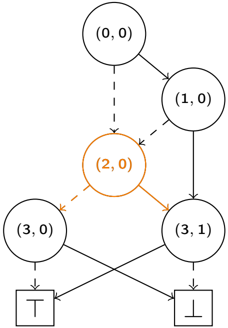
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
$(2, 0)$	0	0

Priority Queue: Q_{count}:

$$
\begin{array}{l}
[\\
\quad ((0,0) \xrightarrow{\perp} (2,0), \quad 1) \quad , \\
\quad ((1,0) \xrightarrow{\perp} (2,0), \quad 1) \quad , \\
\\
\quad ((1,0) \xrightarrow{\top} (3,1), \quad 1) \quad , \\
\quad]
\end{array}
$$

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
(2, 0)	1	0

Priority Queue: Q_{count}:

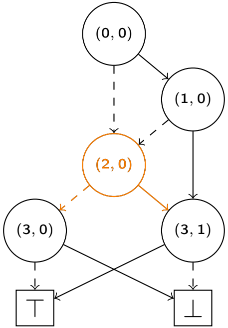
[

$((1,0) \xrightarrow{\perp} (2,0), \quad 1)$,

$((1,0) \xrightarrow{\top} (3,1), \quad 1)$,

]

bdd_pathcount f



(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

Seek	Sum	Result
$(2, 0)$	2	0

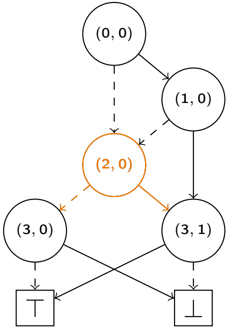
Priority Queue: Q_{count}:

[

$((1, 0) \xrightarrow{\top} (3, 1), \quad 1)$,

]

bdd_pathcount f



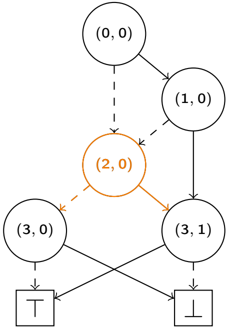
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
(2, 0)	2	0

Priority Queue: Q_{count}:

$$\left[\begin{array}{ll} ((2,0) \xrightarrow{\perp} (3,0), & 2)\ , \\ ((1,0) \xrightarrow{\top} (3,1), & 1)\ , \\ ((2,0) \xrightarrow{\top} (3,1), & 2)\ \end{array}\right]$$

bdd_pathcount f



(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

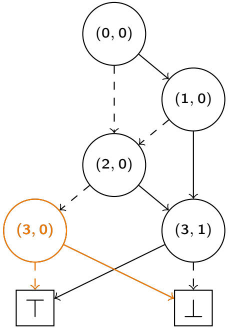
Seek	Sum	Result
$(3,0)$	0	0

Priority Queue: Q_{count}:

$[$

$((2,0) \xrightarrow{\perp} (3,0), \quad 2)$,
$((1,0) \xrightarrow{\top} (3,1), \quad 1)$,
$((2,0) \xrightarrow{\top} (3,1), \quad 2)$ $]$

bdd_pathcount f



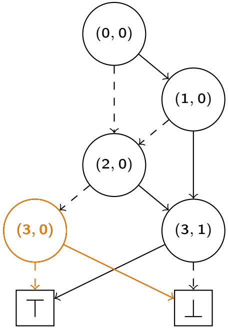
(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

Seek	Sum	Result
$(3, 0)$	0	0

Priority Queue: Q_{count}:

$$[\ ((2,0) \xrightarrow{\perp} (3,0),\ 2),\ ((1,0) \xrightarrow{\top} (3,1),\ 1),\ ((2,0) \xrightarrow{\top} (3,1),\ 2)\ ]$$

bdd_pathcount f



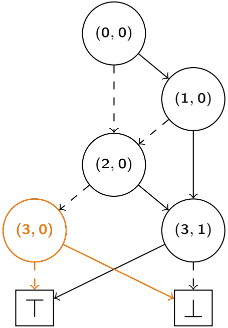
(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

Seek	Sum	Result
$(3, 0)$	2	0

Priority Queue: Q_{count}:

$$[\ ((1,0) \xrightarrow{\top} (3,1),\ 1),\ ((2,0) \xrightarrow{\top} (3,1),\ 2)\ ]$$

bdd_pathcount f



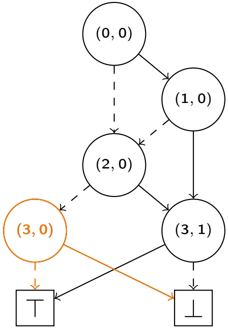
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
$(3, 0)$	2	2

Priority Queue: Q_{count}:

$$[\; ((1,0) \xrightarrow{\top} (3,1),\; 1),\; ((2,0) \xrightarrow{\top} (3,1),\; 2) \;]$$

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
(3, 1)	0	2

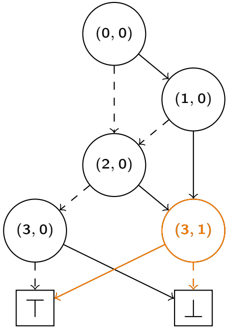
Priority Queue: Q_{count}:

[

$((1,0) \xrightarrow{\top} (3,1), \quad 1)$,

$((2,0) \xrightarrow{\top} (3,1), \quad 2)$]

bdd_pathcount f



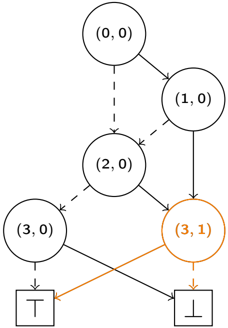
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
(3, 1)	0	2

Priority Queue: Q_{count}:

$$[\; ((1,0) \xrightarrow{\top} (3,1),\ 1),\ ((2,0) \xrightarrow{\top} (3,1),\ 2)\;]$$

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

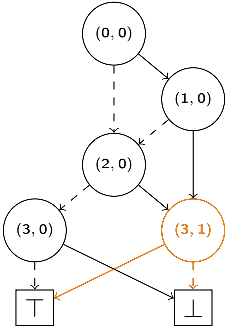
Seek	Sum	Result
(3, 1)	1	2

Priority Queue: Q_{count}:

[

$((2,0) \xrightarrow{\top} (3,1), \quad 2)$]

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

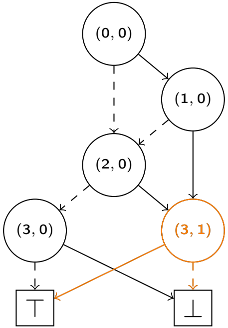
Seek	Sum	Result
(3, 1)	3	2

Priority Queue: Q_{count}:

[

]

bdd_pathcount f



(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

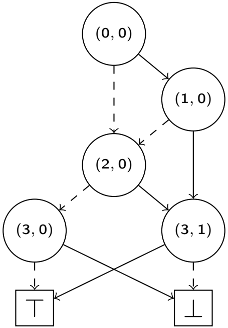
Seek	Sum	Result
$(3, 1)$	3	5

Priority Queue: Q_{count}:

[

]

bdd_pathcount f



(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Result

5

Priority Queue: Q_{count}:

[

]

bdd_satcount f vc

What needs to be changed for a `bdd_satcount f vc`?

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.
Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)

operator (a: Req) < (b: Req) = a.target < b.target
    ∨ (a.target = b.target ∧ a.levels_visited < b.levels_visited)
```

Lemma
Req::$<$ is a total order.

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.
Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
$\#^{\texttt{Req}}\ \mathit{ns}\ (\texttt{Req}\ t\ s\ \ell) \triangleq s \cdot (\#_{\ell}\ \mathit{ns}\ t)$ and $\#^{\texttt{pq}}\ \mathit{ns}\ \mathit{pq} \triangleq \sum_{r \in \mathit{pq}} \#^{\texttt{Req}}\ \mathit{ns}\ r$.

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.
Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
$\#^{\texttt{Req}}\ ns\ (\texttt{Req}\ t\ s\ \ell) \triangleq s \cdot (\#_\ell\ ns\ t)$ and $\#^{\texttt{pq}}\ ns\ pq \triangleq \sum_{r \in pq} \#^{\texttt{Req}}\ ns\ r$.

Lemma
$\#^{pq}\ ns\ \emptyset = 0$

Proof.
Trivial. $\square$

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.
Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
$\#^{\texttt{Req}}\ \mathit{ns}\ (\texttt{Req}\ t\ s\ \ell) \triangleq s \cdot (\#_\ell\ \mathit{ns}\ t)$ and $\#^{\texttt{pq}}\ \mathit{ns}\ \mathit{pq} \triangleq \sum_{r \in pq} \#^{\texttt{Req}}\ \mathit{ns}\ r$.

Lemma
If $\mathtt{pq.top}() = \mathtt{Some}\ r$*, then* $\#^{\mathtt{pq}}\ \mathit{ns}\ \mathtt{pq} = \#^{\mathtt{pq}}\ \mathit{ns}\ \mathtt{pq.pop}() + \#^{\mathtt{Req}}\ \mathit{ns}\ r$

Proof.
Due to $\mathrm{pq} = \{r\} + \mathrm{pq.pop}()$. □

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`. Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
$\#^{\texttt{Req}}\ ns\ (\texttt{Req}\ t\ s\ \ell) \triangleq s \cdot (\#_\ell\ ns\ t)$ and $\#^{\texttt{pq}}\ ns\ pq \triangleq \sum_{r \in pq} \#^{\texttt{Req}}\ ns\ r$.

Lemma
If $\forall r \in pq : r.target \neq i$, then $\#^{pq}\ (i\ t\ e) :: ns = \#^{pq}\ ns\ r$

Proof.
$\#^{\texttt{Req}}\ (i\ t\ e) :: ns\ r = \#^{\texttt{Req}}\ ns\ r$ by definition and some case distinction. $\square$

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.
Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
The priority queue pq is *well formed* wrt. list of nodes *ns* if

- $\{r.\texttt{target} \mid r \in \texttt{pq}\} \subseteq \{i \mid (i\ t\ e) \in ns\}$
- $\forall r \in \texttt{pq} : r.\texttt{levels\_visited} < r.\texttt{target.level}$

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.

Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
The priority queue pq is *well formed* wrt. list of nodes *ns* if

- $\{r.\texttt{target} \mid r \in \texttt{pq}\} \subseteq \{i \mid (i\ t\ e) \in ns\}$
- $\forall r \in \texttt{pq} : r.\texttt{levels\_visited} < r.\texttt{target.level}$

Lemma
An empty priority queue is well formed.

Proof.
Trivial. □

bdd_satcount f vc

Think of a Priority Queue of `Req` as a *Multiset* with (pure) functions `top()` and `pop()`.

Such a data structure can be implemented as a tree!

```
Req = (target : Uid, sum : ℕ, levels_visited : ℕ)
```

Definition
The priority queue pq is *well formed* wrt. list of nodes *ns* if

- $\{r.\texttt{target} \mid r \in \texttt{pq}\} \subseteq \{i \mid (i\ t\ e) \in ns\}$
- $\forall r \in \texttt{pq} : r.\texttt{levels\_visited} < r.\texttt{target.level}$

Lemma
If $\texttt{pq}$ *is well formed then* $\texttt{pq.pop()}$ *is too.*

Proof.
A little bit of set theory. □

bdd_satcount f vc

Accumulate from pqthe sum, sacc, and the number of visited levels, vacc, along all in-going edges to a single node with Uid t.

```
combine_paths' (pq : PQ<Req>) (t : Uid) ((sacc, vacc) : ℕ × ℕ) =
    match pq.top() with
    | None              => (sacc, vacc, pq)
    | Some Req t' s v => if t' = t
                          then let acc' = (sacc·2^(vc−vacc) + s, v)
                                   ; pq'  = pq.pop()
                               in combine_paths' pq' t acc'
                          else (sacc, vacc, pq)

combine_paths (pq : PQ<Req>) (t : Uid) =
    combine_paths' pq t (0,0)
```

bdd_satcount f vc

Accumulate from pqthe sum, `sacc`, and the number of visited levels, `vacc`, along all in-going edges to a single node with `Uid` `t`.

```
combine_paths' (pq : PQ<Req>) (t : Uid) ((sacc, vacc) : ℕ × ℕ) =
    match pq.top() with
    | None              => (sacc, vacc, pq)
    | Some Req t' s v => if t' = t
                         then let acc' = (sacc·2^(vc−vacc) + s, v)
                                  ; pq'  = pq.pop()
                              in combine_paths' pq' t acc'
                         else (sacc, vacc, pq)
```

Lemma
Let $(s', v', pq') = \mathit{combine\_paths}\ pq\ t.\ pq' = \{r \in pq \mid r.\mathit{target} \neq t\}$ *(if* $pq \neq \emptyset$*).*

bdd_satcount f vc

Accumulate from pqthe sum, `sacc`, and the number of visited levels, `vacc`, along all in-going edges to a single node with `Uid` `t`.

```
combine_paths' (pq : PQ<Req>) (t : Uid) ((sacc, vacc) : ℕ × ℕ) =
    match pq.top() with
    | None             => (sacc, vacc, pq)
    | Some Req t' s v => if t' = t
                         then let acc' = (sacc·2^(vc−vacc) + s, v)
                                  ; pq'  = pq.pop()
                              in combine_paths' pq' t acc'
                         else (sacc, vacc, pq)
```

Lemma
Let $(s', v', pq') = \mathit{combine\_paths}\ pq\ t$. *If* pq *is well formed then* pq' *is too.*

bdd_satcount f vc

Accumulate from pqthe sum, `sacc`, and the number of visited levels, `vacc`, along all in-going edges to a single node with `Uid` `t`.

```
combine_paths' (pq : PQ<Req>) (t : Uid) ((sacc, vacc) : ℕ × ℕ) =
    match pq.top() with
    | None           => (sacc, vacc, pq)
    | Some Req t' s v => if t' = t
                        then let acc' = (sacc·2^(vc−vacc) + s, v)
                                 ; pq'  = pq.pop()
                             in combine_paths' pq' t acc'
                        else (sacc, vacc, pq)
```

Lemma
Let $(s', v', pq') = \mathit{combine\_paths}\ pq\ t$. *Then,* $\#^{pq}\ ns\ pq\ t = \#^{pq}\ ns\ pq' + s' \cdot \#_{v'}\ ns\ t$.

bdd_satcount f vc

Forward sum, `s`, and number of visited levels, `l`, along an out-going edge to target `t`.

```
forward_paths (pq : PQ<Req>) (t : Ptr) (s : ℕ) (l : ℕ) =
    match t with
    | Leaf False => (0, pq)
    | Leaf True  => (s·2^(vc−1), pq)
    | Node tgt   => (0, pq.push((t, s, l)))
```

Lemma
Let $(s', pq') = \mathtt{forward\_paths}\ pq\ t\ s\ l$*. If* $l < vc$*,*
then $\#^{pq}\ ns\ pq + s \cdot \#_l\ ns\ t = \#^{pq}\ ns\ pq' + s'$*.*

Proof.
Case analysis and definition of $\#^{\mathtt{pq}}$, $\#^{\mathtt{Req}}$, and $\#_l$. $\square$

bdd_satcount f vc

Forward sum, `s`, and number of visited levels, `l`, along an out-going edge to target `t`.

```
forward_paths (pq : PQ<Req>) (t : Ptr) (s : ℕ) (l : ℕ) =
    match t with
    | Leaf False => (0, pq)
    | Leaf True  => (s·2^(vc−l), pq)
    | Node tgt   => (0, pq.push((t, s, l)))
```

Lemma
Let $(s', pq') = \mathtt{forward\_paths}\ pq\ t\ s\ l$. *If* $t \in ns$ *and* pq *is well formed, then* pq' *is also well formed.*

Proof.
Case analysis and assumptions. □

bdd_satcount f vc

Forward sum, `s`, and number of visited levels, `l`, along an out-going edge to target `t`.

```
forward_paths (pq : PQ<Req>) (t : Ptr) (s : ℕ) (l : ℕ) =
    match t with
    | Leaf False => (0, pq)
    | Leaf True  => (s·2^(vc−l), pq)
    | Node tgt   => (0, pq.push((t, s, l)))
```

Lemma
Let $(s', pq') = \mathtt{forward\_paths}\ pq\ t\ s\ l$*. If* $t = \mathtt{Leaf}\ \_$*, then* $pq' \subseteq pq$*.*
If $t = \mathtt{Node}\ u$*, then* $pq' \subseteq pq + \{(\mathtt{Req}\ u\ s\ l)\}$*.*

Proof.
Case analysis and assumptions. □

bdd_satcount f vc

Accumulate all in-going edges and then forward to children (to-be processed later).

```
bdd_satcount' (ns : List<Node>) (pq : PQ<Req>) (racc : ℕ) =
    match ns , pq.top() with
    | _ , None  => racc
    | n::ns', _ =>
      let (s, lvls, pq') = combine_paths pq n.i
        ; (rt, pq'')     = forward_paths pq'  n.t s (lvls+1)
        ; (re, pq''')    = forward_paths pq'' n.e s (lvls+1)
      in bdd_satcount' ns' pq''' (racc + rt + re)
```

bdd_satcount f vc

Accumulate all in-going edges and then forward to children (to-be processed later).

```
bdd_satcount' (ns : List<Node>) (pq : PQ<Req>) (racc : ℕ) = ...
```

Lemma
Assume $\mathtt{pq}$ *and* ns *are well formed and* $\{n.\mathtt{uid.label} \mid n \in ns\} \subseteq \{0, 1, \ldots \mathtt{vc} - 1\}$*.*
Then, $\mathtt{bdd\_satcount}'\ ns\ \mathtt{pq}\ r = r + \#^{\mathtt{pq}}\ ns\ \mathtt{pq}$*.*

Proof.
Induction in *ns* and case analysis on top of `pq`. Use previous lemmata to skip node (if not the target) or to parse correctness through `combine_paths` and `forward_paths`.

To this end, one needs to bound the number of visited levels in each request by `vc`. Furthermore, the results `r`, `r'`, and `r''` are combined with the lemmata for $\#_n$ (Appendix). $\square$

bdd_satcount f vc

Finally, deal with the root for a BDD f.

```
bdd_satcount (f : Bdd) (vc : ℕ) =
    match f with
    | Leaf  False => 0
    | Leaf  True  => 2^vc
    | Nodes r::ns =>
        let pq          = ∅
          ; (rt, pq')  = forward_paths pq  r.t 1 1
          ; (re, pq'') = forward_paths pq' r.e 1 1
        in bdd_satcount' ns ∅ (rt + re)
```

bdd_satcount f vc

Finally, deal with the root for a BDD f.

```
bdd_satcount (f : Bdd) (vc : ℕ) = ...
```

Theorem
If f is well formed (incl. $\{n.\mathtt{uid.label} \mid n \in \mathtt{Bdd::Nodes} ns\} \subseteq \{0, 1, \ldots vc - 1\}$),
then $\mathtt{bdd\_satcount}\ f\ vc = \#_0\ f$.

Proof.
Leaf cases are trivial. For nodes *ns*, use lemmata for `forward_paths` to prove preconditions for `bdd_satcount'` *ns* `pq` $(0, 0)$ correctness. □

Take Home Message...

- (Some) Time-forward processing algorithms can be implemented *functionally*.
  - They are *pure* and *tail-recursive*.
  - They are I/O-efficient since they only work on lists and trees [Blelloch & Harper].
- Proving correctness is feasible (see also github.com/SSoelvsten/cadiar)
  - Further refinment possible to get closer to the C++ performance.
- One can prove time and I/O complexity.

Steffan Christ Sølvsten

`soelvsten@cs.au.dk`

ssoelvsten.github.io

Adiar

github.com/ssoelvsten/adiar

ssoelvsten.github.io/adiar

AARHUS UNIVERSITY

Contents

Motivation

Correctness of Time-forward Processing

- Encoding Binary Decision Diagrams
- bbd_eval
- bbd_not
- bbd_satcount

Appendix

- $\lfloor a \rfloor_n$: Bounded Domain
- $\#_n$: Number of Assignments

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition

$\lfloor a \rfloor_n \triangleq (a\,\overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\perp\})$.

Lemma (alternative definition)
$\lfloor a \rfloor_n \iff \forall i : i \notin \{n, n+1, \ldots, vc - 1\} \implies a\ i = \perp.$

Proof.
From definition. $\square$

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma
If $n > vc$, then $\lfloor a \rfloor_n \iff a = \lambda\_.\bot$.

Proof.
From (alternative) definition. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\perp\})$.

Lemma
For all $n \in N$, $\lfloor a \rfloor_n \wedge \neg a\ n \iff \lfloor a \rfloor_{n+1}$.

Proof.
Case analysis of $i = n$ and (alternative) definition. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma
For all $n \in N$, $\lfloor a \rfloor_n \iff (\lfloor a \rfloor_n \wedge a\ n) \vee \lfloor a \rfloor_{n+1}$.

Proof.
Case analysis of $i = n$ and (alternative) definition. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition

$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma

$\{a \mid \lfloor a \rfloor_n\}$ *is finite.*

Proof.

Induction in n and some set theory. $\square$

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma
If $n < vc$, $|\{a \mid \lfloor a \rfloor_n \wedge a\ n\}| = |\{a \mid \lfloor a \rfloor_{n+1}\}|$.

Proof.
Previous lemmas together with set theory. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma
If $n < vc$, $|\{a \mid \lfloor a \rfloor_{vc-n}\}| = 2^n$ and $|\{a \mid \lfloor a \rfloor_n\}| = 2^{vc-n}$.

Proof.
Induction in n, case analysis on `dom_bounded` with a and with $\mathtt{vc} - n$ and $\mathtt{vc} - (n+1)$, respectively, and some set theory. $\square$

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\perp\})$.

Lemma
If $\lfloor a \rfloor_n$*, then* $\lfloor (a[x := \perp]) \rfloor_n$.

Proof.
From (alternative) definition. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition

$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma

If $n \leq x \leq \mathtt{vc}$, then $\lfloor a \rfloor_n \iff \lfloor a[x := \mathbb{B}] \rfloor_n$.

Proof.

From (alternative) definition. $\square$

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, vc, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathrm{vc}\}} \subseteq \{\bot\})$.

Lemma
If $x < t.label$*, then* `bdd_eval'` *ns t* $a[x := \mathbb{B}] =$ `bdd_eval'` *ns t a.*

Proof.
Induction in *ns* with a case analysis of the algorithms branches. Here, use that the BDD is *well formed*. □

Bounded Domain

For this, we need to work with Boolean functions with a bounded domain.

Let the *variable count*, $\mathtt{vc}$, be fixed.

Definition
$\lfloor a \rfloor_n \triangleq (a\ \overline{\{i \mid n \leq i < \mathtt{vc}\}} \subseteq \{\bot\})$.

Lemma
$|\{a \mid \neg a\ x \wedge \textit{bdd\_eval'}\ ns\ t\ a \wedge \lfloor a \rfloor_n\}| = |\{a \mid \textit{bdd\_eval'}\ ns\ t\ a \wedge \lfloor a \rfloor_{n+1}\}|$.

Proof.
Set theory and previous lemma $\mathtt{bdd\_eval'}$ $ns\ t\ (a[x := \mathbb{B}])$. □

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|.$

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
If $n \leq vc$, $\#_n\ \top = 2^{vc-n}$.

Proof.
From definition of `bdd_eval` and lemma on $\lfloor a \rfloor_n$. □

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
$\#_n \perp = 0$.

Proof.
From definition of `bdd_eval`. □

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
$\#_n\ f = 2^{n-|\mathit{support}(f)|} \cdot \#_{\mathit{support}(f)}\ f$.

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
$\#_n\ \mathit{Node}(i\ t\ e) :: ns = \#^t_{n+1}\ ns + \#^t_{n+1}\ ns$

Proof.
Use *well formedness*, *set theory*, and previous lemmata about $\lfloor a \rfloor_n$ to prove:

- $a\ i \implies \texttt{bdd\_eval'}\ ns\ t\ a \wedge \lfloor a \rfloor_n$ and $\neg a\ i \implies \texttt{bdd\_eval'}\ ns\ e\ a \wedge \lfloor a \rfloor_n$.
- Can split set of assignments S into $S_t \cup S_e$.
- $S_t = \{a \mid a\ x \wedge \texttt{bdd\_eval'}\ ns\ e\ a \wedge \lfloor a \rfloor_{n+1}\}$ and $S_e = \{a \mid \neg a\ x \wedge \texttt{bdd\_eval'}\ ns\ t\ a \wedge \lfloor a \rfloor_{n+1}\}$.
- $S_t \cap S_e = \emptyset$ and hence $|S| = |S_t| + |S_e|$.

□

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
If $n.uid < t$, then $\texttt{bdd\_eval'}\ n :: ns\ t\ a = \texttt{bdd\_eval'}\ ns\ t\ a$.

Proof.
Simple case analysis. $\square$

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
$\#_n\ (i\ t\ e) :: ns\ t = \#_n\ ns\ t$ *and* $\#_n\ n :: ns\ e = \#_n\ ns\ e$.

Proof.
Due to levels are *strictly* increasing in BDDs. $\square$

Number of Assignments

We need a mathematical notion of the number of assignments satisfying a BDD.

Definition
$\#_n\ f \triangleq |\{a \mid \texttt{bdd\_eval}\ f\ a \wedge \lfloor a \rfloor_n\}|$.

Lemma
If $i \neq u$, then $\#_n\ (i\ t\ e) :: ns\ u = \#_n\ ns\ u$.

Proof.
By definition. $\square$