Correctness of Time-forward Processing Algorithms

Steffan Sølvsten, Simon Wimmer

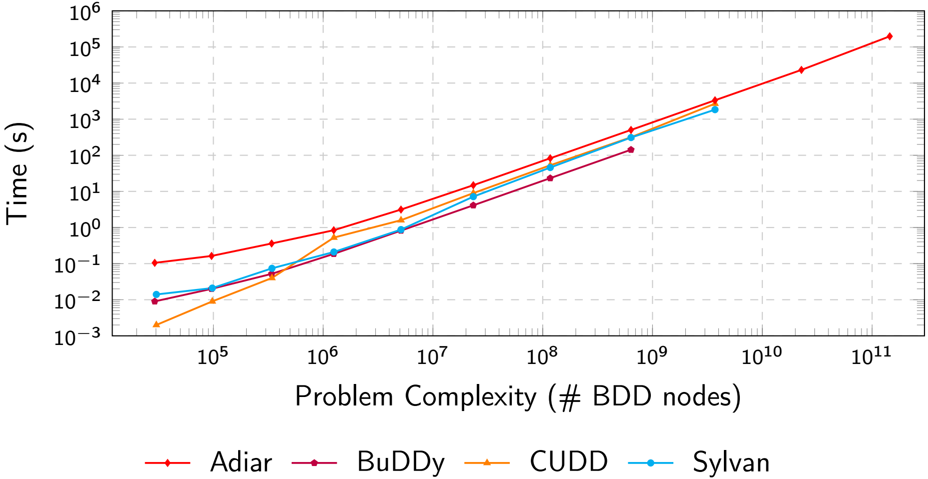
LogSem Seminar, 2nd of December 2024

AARHUS
UNIVERSITY

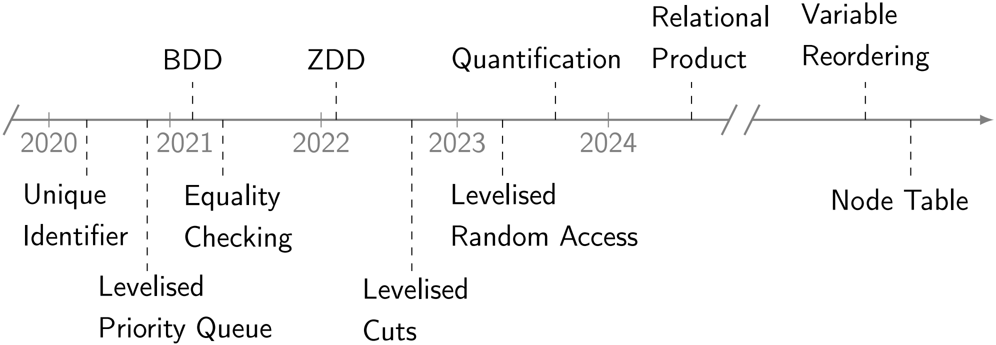
Adiar

I/O-efficient Decision Diagrams

github.com/logsem/adiar



Running Time to solve *N-Queens* problems.


BDD
ZDD
Quantification
Relational Product
Variable Reordering
2020
2021
2022
2023
2024
Unique Identifier
Levelised Priority Queue
Equality Checking
Levelised Cuts
Levelised Random Access
Node Table

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 uint64 shifted_negate =
    ((uint64) negate) << ptr_uint64::data_shift;
  return p.is_leaf() ? 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

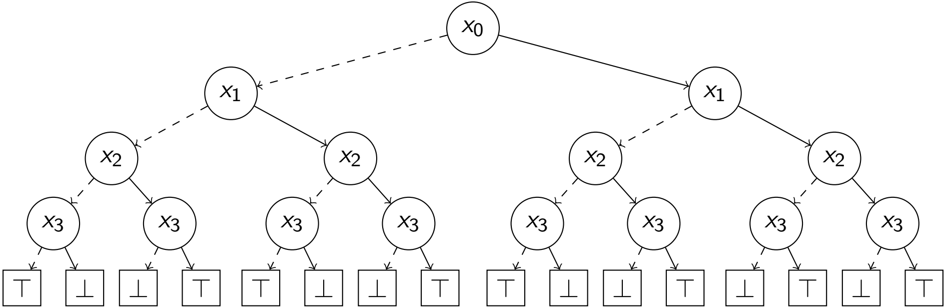
Contents

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)

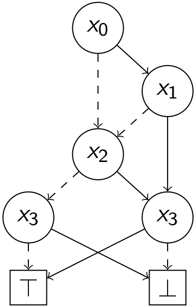
Semantics of BDDs



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)

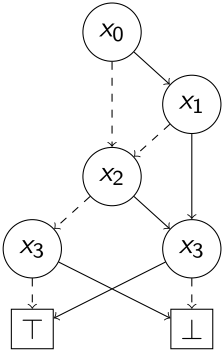
Data Types: Unique Identifiers and Pointers

```
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
$\mathtt{Uid}.<$ and $\mathtt{Ptr}.<$ are total orders.

Proof.
Trival case distinctions. $\square$

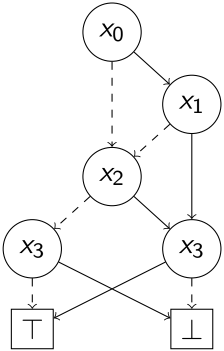
Data Types: Nodes and BDDs



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

```
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.i`.

Contents

bdd_eval f x

```
bdd_eval' ((i t e)::ns': List[Node]) (tgt: Uid) (x: ℕ → 𝔹) =
     if i < tgt
     then bdd_eval' ns' tgt x
     else case if x(a) then t else e of
               Leaf(b)     → b
               Node(tgt') → bdd_eval' ns' tgt' x

bdd_eval (val   : Bdd.Leaf)  (x: ℕ → 𝔹) = val
bdd_eval (r::ns : Bdd.Nodes) (x: ℕ → 𝔹) = bdd_eval' r::ns r x
```

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.

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.

Theorem
If f is well formed, then $\forall x$: $\texttt{bdd\_eval}\ f\ x \iff \texttt{bdt\_eval}\ (\texttt{bdt\_of\_bdd}\ f)\ x$.

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

bdd_not f

```
operator ! (p: Ptr) = case p of Leaf v → Leaf !v
                                Node u → Node u

bdd_not (val : Bdd.Leaf)  = Bdd.Leaf !v
bdd_not (ns  : Bdd.Nodes) = Bdd.Nodes (map λ(i t e) → (i !t !e) ns)
```

bdd_not f

```
operator ! (p: Ptr) = case p of Leaf v → Leaf !v
                                Node u → Node u

bdd_not (val : Bdd.Leaf)  = Bdd.Leaf !v
bdd_not (ns  : Bdd.Nodes) = Bdd.Nodes (map λ(i t e) → (i !t !e) ns)
```

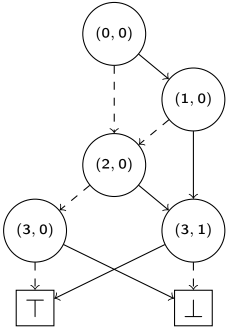
Theorem
If f is well formed, then $\forall x$: $\neg$ `(bdd_eval f x)` $\iff$ `bdd_eval (bdd_not f) x`.

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

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

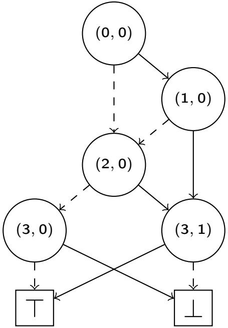
Contents

bdd_pathcount f



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

bdd_pathcount f



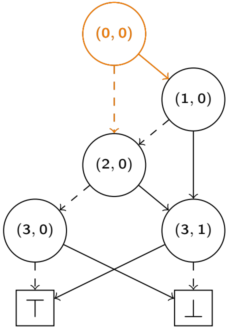
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

[

]

bdd_pathcount f



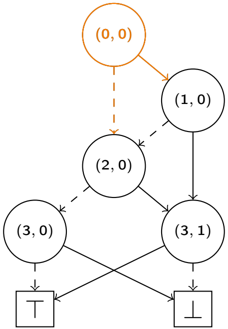
(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

[

]

bdd_pathcount f

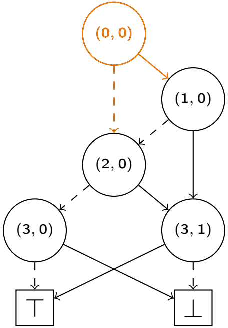


(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Priority Queue: Q_{count}:

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

bdd_pathcount f



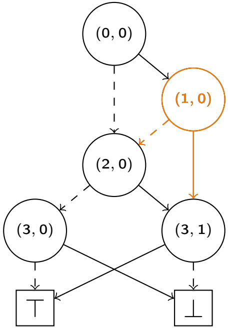
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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

Priority Queue: Q_{count}:

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

bdd_pathcount f



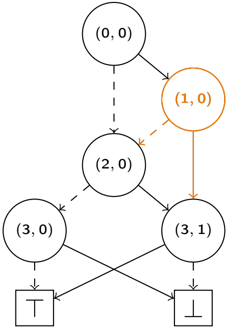
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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

Priority Queue: Q_{count}:

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

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

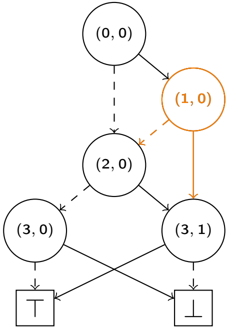
Priority Queue: Q_{count}:

[

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

]

bdd_pathcount f



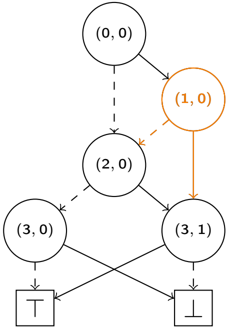
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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

Priority Queue: Q_{count}:

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

bdd_pathcount f



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

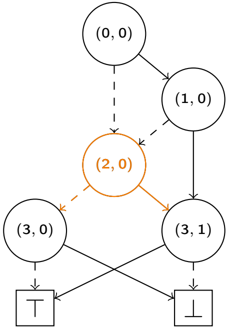
Seek	Sum	Result
(2, 0)	0	0

Priority Queue: Q_{count}:

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

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

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}:

[

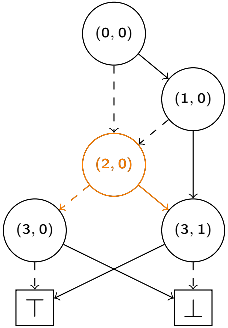
$((0,0) \xrightarrow{\perp} (2,0), \quad 1)$,

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

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

]

bdd_pathcount f



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

Seek	Sum	Result
$(2, 0)$	1	0

Priority Queue: Q_{count}:

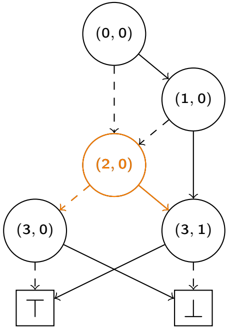
[

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

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

]

bdd_pathcount f



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

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

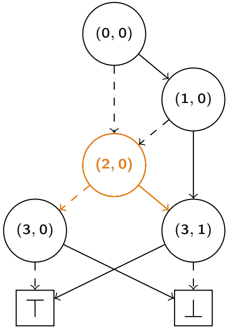
Priority Queue: Q_{count}:

[

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

]

bdd_pathcount f



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

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

Priority Queue: Q_{count}:

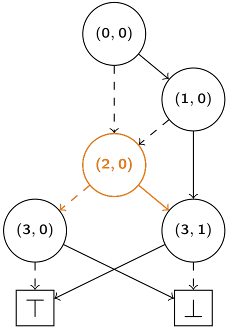
$[$

$((2,0) \xrightarrow{\bot} (3,0), \quad 2) \quad ,$

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

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

bdd_pathcount f



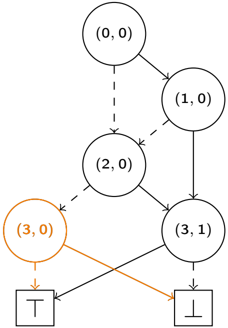
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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



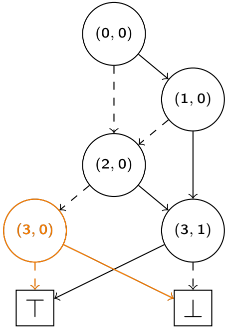
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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

Priority Queue: Q_{count}:

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

bdd_pathcount f



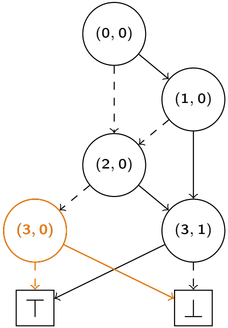
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
(3, 0)	2	0

Priority Queue: Q_{count}:

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

bdd_pathcount f



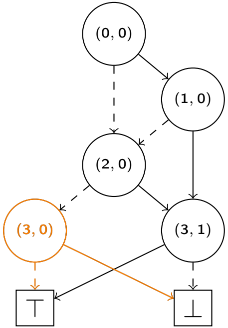
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

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

Priority Queue: Q_{count}:

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

bdd_pathcount f



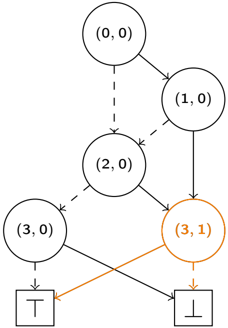
(a) $(x_0 \land x_1 \land x_3) \lor (x_2 \oplus x_3)$

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



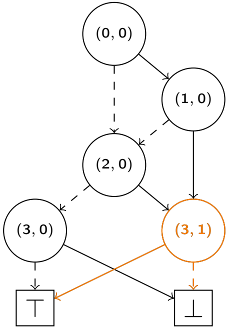
(a) $(x_0 \wedge x_1 \wedge x_3) \vee (x_2 \oplus x_3)$

Seek	Sum	Result
$(3, 1)$	0	2

Priority Queue: Q_{count}:

$$
\begin{aligned}
[\; & ((1,0) \xrightarrow{\top} (3,1), \quad 1) \quad , \\
& ((2,0) \xrightarrow{\top} (3,1), \quad 2) \quad]
\end{aligned}
$$

bdd_pathcount f



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

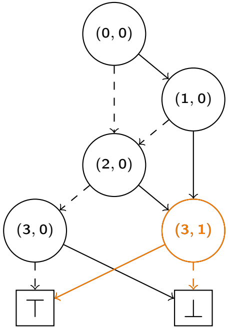
Seek	Sum	Result
$(3, 1)$	1	2

Priority Queue: Q_{count}:

[

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

bdd_pathcount f



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

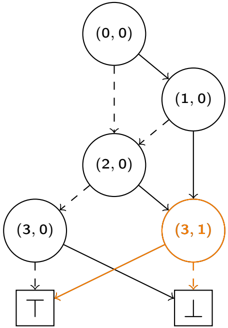
Seek	Sum	Result
(3, 1)	3	2

Priority Queue: Q_{count}:

[

]

bdd_pathcount f



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

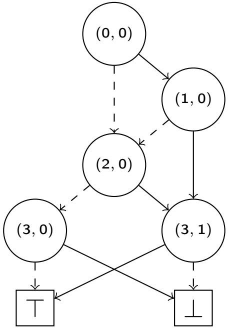
Seek	Sum	Result
(3, 1)	3	5

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}:

[

]

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()`.

```
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 partial order.

Proof.
Trivial case distinctions. □

bdd_satcount f vc

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

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

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

bdd_satcount f vc

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

```
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()`.

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

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

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

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

bdd_satcount f vc

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

```
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* $\#^{pq}\ (i\ t\ e) :: ns\ pq = \#^{pq}\ ns\ pq$

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

bdd_satcount f vc

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

```
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()`.

```
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()`.

```
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 pq is well formed then $pq.pop()$ is too.

Proof.
A little bit of set theory. □

bdd_satcount f vc

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

```
combine_paths' (pq : PQ<Req>) (tgt : Uid) ((sacc, lacc) : ℕ × ℕ) =
    case pq.top() of
      None               → (sacc, lacc, pq)
      Some Req tgt' s ℓ → if tgt' ≠ tgt
                           then (sacc, lacc, pq)
                           else let acc' = (sacc·2^(ℓ−lacc) + s, ℓ)
                                  ; pq'  = pq.pop()
                                in combine_paths' pq' tgt acc'

combine_paths (pq : PQ<Req>) (tgt : Uid) =
    combine_paths' pq tgt (0,0)
```

bdd_satcount f vc

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

```
combine_paths' (pq : PQ<Req>) (tgt : Uid) ((sacc, lacc) : ℕ × ℕ) =
    case pq.top() of
      None               → (sacc, lacc, pq)
      Some Req tgt' s ℓ → if tgt' ≠ tgt
                            then (sacc, lacc, pq)
                            else let acc' = (sacc·2^(ℓ−lacc) + s, ℓ)
                                   ; pq'  = pq.pop()
                                 in combine_paths' pq' tgt acc'
```

Lemma
Let $(s', \ell', pq') = \texttt{combine\_paths}\ pq\ t$*, if* $\forall r \in pq : r.target \geq i$*, then* $pq' = \{r \in pq \mid r.target \neq t\}$*.*

bdd_satcount f vc

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

```
combine_paths' (pq : PQ<Req>) (tgt : Uid) ((sacc, lacc) : ℕ × ℕ) =
    case pq.top() of
      None               → (sacc, lacc, pq)
      Some Req tgt' s ℓ → if tgt' ≠ tgt
                            then (sacc, lacc, pq)
                            else let acc' = (sacc·2^(ℓ−lacc) + s, ℓ)
                                     ; pq'  = pq.pop()
                                 in combine_paths' pq' tgt acc'
```

Lemma

Let $(s', \ell', pq') = \texttt{combine\_paths}\ pq\ t$, if pq is well formed then pq' is too.

bdd_satcount f vc

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

```
combine_paths' (pq : PQ<Req>) (tgt : Uid) ((sacc, lacc) : ℕ × ℕ) =
    case pq.top() of
      None              → (sacc, lacc, pq)
      Some Req tgt' s ℓ → if tgt' ≠ tgt
                          then (sacc, lacc, pq)
                          else let acc' = (sacc·2^(ℓ−lacc) + s, ℓ)
                                   ; pq'  = pq.pop()
                               in combine_paths' pq' tgt acc'
```

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

bdd_satcount f vc

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

```
forward_paths (pq : PQ<Req>) (ptr : Ptr) (s : ℕ) (ℓ : ℕ) =
    case s, ptr of
      0, _             → (0, pq) (* Well formed ⇏ fully connected *)
      _, Leaf False → (0, pq)
      _, Leaf True   → (s·2^(vc−ℓ), pq)
      _, Node tgt    → (0, pq + {(tgt, s, ℓ)})
```

bdd_satcount f vc

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

```
forward_paths (pq : PQ<Req>) (ptr : Ptr) (s : ℕ) (ℓ : ℕ) =
    case s, ptr of
      0, _            → (0, pq) (* Well formed ⇏ fully connected *)
      _, Leaf False → (0, pq)
      _, Leaf True   → (s·2^(vc−ℓ), pq)
      _, Node tgt    → (0, pq + {(tgt, s, ℓ)})
```

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

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

bdd_satcount f vc

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

```
forward_paths (pq : PQ<Req>) (ptr : Ptr) (s : ℕ) (ℓ : ℕ) =
    case s, ptr of
      0, _           → (0, pq) (* Well formed ⇏ fully connected *)
      _, Leaf False → (0, pq)
      _, Leaf True  → (s·2^(vc−ℓ), pq)
      _, Node tgt   → (0, pq + {(tgt, s, ℓ)})
```

Lemma
Let $(s', pq') = \mathit{forward\_paths}\ pq\ t\ s\ \ell$. *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 ptr.

```
forward_paths (pq : PQ<Req>) (ptr : Ptr) (s : ℕ) (ℓ : ℕ) =
    case s, ptr of
      0, _            → (0, pq) (* Well formed ⇏ fully connected *)
      _, Leaf False → (0, pq)
      _, Leaf True  → (s·2^(vc−ℓ), pq)
      _, Node tgt   → (0, pq + {(tgt, s, ℓ)})
```

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

Proof.
Case analysis and assumptions. $\square$

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) =
    case ns    , pq.top() of
      _      , None  → racc
      n::ns', _      →
        let (s, ℓ, pq') = combine_paths pq   n.i
          ; (rt, pq'')  = forward_paths pq'  n.t s (ℓ+1)
          ; (re, pq''') = forward_paths pq'' n.e s (ℓ+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* $\{n.\mathtt{uid.level} \mid n \in ns\} \subseteq \{0, 1, \ldots \mathtt{vc} - 1\}$.
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, `racc`, `rt`, and `re`, are combined with the lemmata for $\#_n$ (Appendix). □

bdd_satcount f vc

Finally, deal with the root for a BDD f.

```
bdd_satcount (False : Bdd.Leaf)  (vc : ℕ) = 0
bdd_satcount (True  : Bdd.Leaf)  (vc : ℕ) = 2^vc
bdd_satcount (r::ns : Bdd.Nodes) (vc : ℕ) =
    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 (   _    : Bdd)        (vc : ℕ) = ...
```

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

Proof.
Leaf cases are trivial. For nodes ns, use lemmata for `forward_paths` to prove preconditions for $\texttt{bdd\_satcount}'\ ns\ \texttt{pq}\ (0, 0)$ correctness. □

Take Home Message...

- 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 them to be efficient both with respect to 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

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

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

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. $\square$

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
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. $\square$

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
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. $\square$

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
$\{a \mid \lfloor a \rfloor_n\}$ *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

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

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. $\square$

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 $n < vc$, $|\{a \mid \lfloor a \rfloor_{vc-n}\}| = 2^n$ 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. □

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

Proof.
From (alternative) definition. $\square$

Bounded Domain

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

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

Definition

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

Lemma

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

Proof.

From (alternative) definition. □

Bounded Domain

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

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

Definition

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

Lemma

If $x <$ *t.level, 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*, vc, be fixed.

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

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

Proof.
Set theory and previous lemma `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 $n \leq \upsilon c$, $\#_n\ \top = 2^{\upsilon c - n}$.

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

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
$\#_n\ \bot = 0$.

Proof.
From definition of `bdd_eval`. $\square$

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

Number of Assignments

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

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

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:

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

$\square$

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

Proof.
Simple case analysis. □

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. □

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. □