Verified Sentential Decision Diagrams

Steven Holtzen
Saketh Ram Kasibatla

March 15, 2017

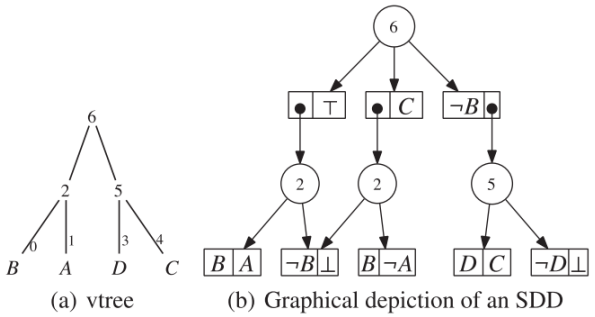

Figure 1: An SDD for the Boolean formula $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. Reproduced from [**?**].

1 Introduction and Motivation

The goal of this project is to construct and reason about sentential decision diagrams (SDDs) in Coq. A sentential decision diagram is a data structure which encodes an arbitrary Boolean formula; see Fig.1 for an example SDD.

The SDD data structure has several desirable properties:

nenumerate

Canonicity. Any two equivalent Boolean formulas will always compile to the same SDD.

Determinism. Any given Or-node in an SDD will have at most one true child.

Decomposability. The children of any given And-node in an SDD share no variables.

These properties allow one to perform many queries in linear time the SDD data structure, for example: model enumeration, weighted model counting, satisfiability and validity checking. In our work, we focused on non-canonical (i.e. uncompressed) SDDs, which are easier to represent and manipulate.

The key operations that we focused on implementing were (1) SDD compilation; (2) SDD application; (3) SDD evaluation.

1.1 Proof Goals

Our original goal was to prove the following properties:

1. Correctness: For all Boolean formalae e and inputs i, `eval_boolexpr`(e, i) = `eval_sdd`(`compile`(e), i).
2. Prove that the resulting SDD is decomposable.
3. Prove that the resulting SDD is deterministic.
4. Prove that `init_var` generates SDDs with respect to the same v-tree (will most likely be a necessary lemma for proving the correctness of `apply`).

During the course of implementation, these goals proved to be too ambitious. We were able to prove, subject to certain assumptions, that SDDs remain deterministic after application. We relaxed goal (1) to be to prove a particular property of the SDD post-appliction

The proof of this fact required a custom inductive hypothesis on the derivation of `sdd_apply`, which we

```
Theorem apply_unsat :
  forall sdd1 sdd2 sddres,
    sdd_unsat sdd1 ->
    sdd_apply OAnd sdd1 sdd2 sddres ->
    sdd_unsat sddres.
```

Figure 2: The apply unsat theorem.

will elaborate on in later sections. The reason that this approach can not be extended to prove stronger claims and possbile future work will also be discussed.

2 Approach

2.1 OCaml Prototype and Fixpoints vs. Inductive Constructions

We created an initial OCaml implementation of SDD compilation and application to guide our Coq implementation. This implementation can be found in the `ml/` directory of the project.

Some key observations of this OCaml code is that it is reasonably complex; a direct translation of this code to Gallina was attempted, but ultimately we decided against it for the following reasons.

- It was difficult to establish that all the recursive calls (especially `apply`) provably terminated.
- Gallina did not provide as strong of a structure for proofs (improve this)

2.2 Inductive SDD Data Structure

We used the following compact definition for our SDD datastructure:

```
Inductive atom : Type :=
| AFalse : atom
| ATrue : atom
| AVar :  nat -> bool -> atom.
Inductive sdd : Type :=
| Or: list (sdd * sdd) -> sdd
| Atom : atom -> sdd.
```

An `or` node is simply a list of pairs (p_i, s_i) (primes and subs).

2.3 SDD Application

SDD application is the process of combining two SDDs α and β into a third SDD γ according to some operation $\circ$. We defined application as an 4-argument inductive relation:

```
Inductive sdd_apply : op -> sdd -> sdd -> sdd -> Prop
```

where `op` is either $\wedge$ or $\vee$. We decomposed this relation in 3 mutually recursive sub-relations:

1. `atom_apply`, which handles applying together atoms
2. `apply_or_list`, two SDD or-nodes as arguments and produces a new SDD `or`-node.
3. `apply_single_list`, which takes a prime p, a sub s, and an `or`-node $[(p_i, s_i)]$ as an argument and produces the SDD of the form $[(p_i \wedge p, s_i \circ s)]$ for all i, where $\circ$ is applying `op`.

   For this operation, if $p_i \wedge p$ is not satisfiable, it is not to be included in the final `or`-node which is produced. This necessitates an `sdd_sat` and `sdd_unsat` relation.

2.4 Custom Inductive Hypothesis

Ultimately in order to prove theorems

2.5 SDD V-Tree

We experimented with (1) representing the V-Tree as a dependent type of the SDD and (2) creating a seperate `sdd_vtree : sdd -> vtree` inductive relation which generates a V-Tree from a particular SDD:

```
Inductive sdd_vtree : sdd -> vtree -> Prop :=
| AtomTrue : forall n,
 sdd_vtree (Atom ATrue) (VAtom n)
| AtomFalse : forall n,
 sdd_vtree (Atom AFalse) (VAtom n)
| AtomVar : forall n b,
 sdd_vtree (Atom (AVar n b)) (VAtom n)
| OrEmpty : forall v, sdd_vtree (Or []) v
| OrSingle: forall prime sub lvtree rvtree tail,
 sdd_vtree prime lvtree ->
```

```
sdd_vtree sub rvtree ->
sdd_vtree (Or (tail)) (VNode lvtree rvtree) ->
sdd_vtree (Or ((prime, sub) :: tail))
 (VNode lvtree rvtree)
```

2.6 SDD Compilation

SDD compilation is the process of transforming a Boolean expression into an SDD. This procedure requires the following components:

1. An implementation of Boolean expressions.
2. A method which generates an

2.7 Fixed-Width SDD

2.8 Custom Inductive Hypotheses