Verification of BDD Normalization

Veronika Ortner and Norbert Schirmer

Technische Universität München, Institut für Informatik
{ortner|schirmer}@in.tum.de

Abstract. We present the verification of the normalization of a binary decision diagram (BDD). The normalization follows the original algorithm presented by Bryant in 1986 and transforms an ordered BDD in a reduced, ordered and shared BDD. The verification is based on Hoare logics and is carried out in the theorem prover Isabelle/HOL. The work is both a case study for verification of procedures on a complex pointer structure, as well as interesting on its own, since it is the first proof of functional correctness of the pointer based normalization process we are aware of.

1 Introduction

Binary Decision Diagrams (BDDs) are a canonical, memory efficient pointer structure to represent boolean functions, with a wide spread application in computer science. They had a decisive impact on scaling up the technology of model checking to large state-spaces to handle practical applications [5]. BDDs were introduced by Bryant [3], and later on their implementation was refined [2]. The efficiency of the BDD algorithms stems from the sharing of subgraphs in the BDD. Some properties of the BDD, like the uniqueness of the representation of a function, rely on the fact that the graph is maximally shared. So the algorithms manipulating BDDs have to ensure this invariant. However the formal verification of the algorithms has never received great attention. We are only aware of the work of Verma et al. [14] and of Kristic and Matthews [7]. Maybe one reason for the lack of formal verification is the problem of reasoning about imperative pointer programs in general, which is still an area of active research, recently in two main directions: The integration and mechanization of pointer verification in theorem provers [1,8,6], and the development of a new logic, namely separation logic [12]. In this context our work contributes to two aspects. On the one hand it presents the first formal verification of the pointer based normalization algorithm for BDDs as presented by Bryant. Verma et al. [14] and also Kristic and Matthews [7] use a different (more abstract) model of BDDs, where the normalization algorithm is no issue, since they ensure that only normalized BDDs will be constructed at all. Although modern BDD packages also follow this approach, and avoid the costly normalization process, the concepts we introduce to formally describe the invariants on the BDD pointer structure can also serve as basis in a more involved setting like [2]. On the other hand this work is a case study on the feasibility of pointer verification based on Hoare logics in a theorem

prover. It carries on the approach of [8] to (recursive) procedures. In contrast to separation logic, which is difficult to combine with existing theorem proving infrastructure, our embedding of Hoare logics fits seamlessly into Isabelle/HOL.

The rest of the paper is structured as follows. In Sect. 2 we give a short introduction to Isabelle/HOL in general and the Hoare logic module, that we will use for our verification work. Sect. 3 gives an informal overview of BDDs and Sect. 4 introduces our formalization of them. Sect. 5 is devoted to the normalization of BDDs, where we explain the algorithm and describe the assertions and invariants that we have used for the correctness proof. Finally Sect. 6 concludes.

2 Preliminaries

2.1 Isabelle

Isabelle/HOL [10] is an interactive theorem prover for HOL, higher order logic, with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions. Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of Isabelle's nonstandard notation.

There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for function space. To emulate partial functions the polymorphic option type is frequently used: **datatype** *'a option* = *None* | *Some 'a*. Here *'a* is a type variable, *None* stands for the undefined value and *Some x* for a defined value *x*. A partial function from type T_1 to type T_2 can be modelled as $T_1 \Rightarrow (T_2$ *option*). Lists (type *'a list*) come with the empty list [], the infix constructor # and the infix @ that appends two lists, and the conversion function *set* from lists to sets. The nth element of a list *xs* is obtained by *xs* ! *n*. The standard function *map* is also available.

2.2 The Hoare Logic Module

Before considering the algorithm of BDD normalization in detail, we first take a brief look at the verification environment for imperative programs [13] built on top of Isabelle/HOL. It embeds a sequential imperative programming language with (mutually) recursive procedures, local and global variables and a heap for pointer structures into Isabelle/HOL. The expressions used in the language are modelled as ordinary HOL expressions. Therefore both "pseudo code" and C programs can be expressed in a uniform framework, giving the user the freedom to choose the proper abstraction level.

To define the state-space, we use **record**s in Isabelle/HOL [10], which contain every program variable used in the implementation. We can refer to these state-space components by specifying $\acute{v}$ for the current state of the component v and ${}^{\sigma}v$ for the same component at a fixed state σ. Assertions are sets of states and we provide special brackets $\{\!|\ |\!\}$ for them, e.g. $\{\!|\, M = 2 \,|\!\}$ is a shorthand for

the ordinary set comprehension $\{s \mid M\ s = 2\}$, which is the set of states where variable M is equal to 2. M is a record selector of the state-space.

A judgement in our Hoare logic is of the general form $\Gamma,\Theta\vdash P\ c\ Q$ for partial correctness and $\Gamma,\Theta\vdash_t P\ c\ Q$ for total correctness, where P and Q figure as pre- and postcondition. The two remaining variables are premises of the Hoare triple, Γ being the procedure environment and context Θ representing a set of Hoare triples that we may assume. The procedure environment maps procedure names to their bodies. The Hoare triples in Θ are important for proving recursive procedures. An empty set of assumptions can be omitted.

Moreover the module supplies a verification condition generator built on top of the Hoare logic for the programming language.

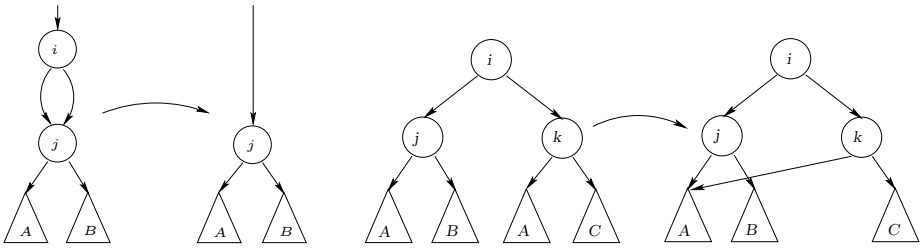
3 Binary Decision Diagrams

"Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions." ([3], p. 1) Thus the representation of Boolean functions by an efficient data-structure is of high interest for computer science. *Binary Decision Diagrams*, which represent the underlying binary decision tree of a Boolean function as a directed acyclic graph (DAG), save storage space and computation time by eliminating redundancy from the canonical representations. Besides, using *reduced*, *ordered* and *shared* BDDs allows us to provide a unique representation for each function. An inner node of the BDD contains a variable over which the Boolean function is defined, together with a pointer to the left and right sub-BDD. Given a valuation of the variables the value of the Boolean function encoded in the BDD is obtained by traversing the BDD according to the valuation of the variables. The leaf that we reach holds the value of the function under the given valuation.

As BDDs are an efficient representation for Boolean functions, which are used in a lot of domains of computer science, there is a wide variety of imaginable operations on them. We will only treat the normalization here, which has an ordered BDD as its argument and removes all redundancies contained in it. The result is an ordered, reduced and shared BDD implementing the same Boolean function. The normalization follows the algorithm presented in [3], where it is used as a central building block for further BDD-algorithms.

The basic transformations on the BDD that occur during normalization are *reducing* and *sharing* (Fig. 1). A node is *reduced* from the BDD if it is irrelevant for the encoded Boolean function: if the left and the right child both point to the same node, it is irrelevant if we choose to go left or right during evaluation. Two sub-BDDs are *shared* if they contain the same decision tree. Note that sharing does not change the structure of the underlying decision tree, but only the structure of the DAG, whereas reducing also changes the decision tree. None of the transformations change the encoded Boolean function.

BDDs are an extreme area of application for pointer programs with operations involving side effects due to the high degree of sharing in the data-structure.



(a) Reducing (b) Sharing

Fig. 1. Illustration of normalization operations

Because of their efficiency in computation time and storage, they are highly popular for the processing of Boolean functions. Altogether BDDs constitute a perfect domain for a case study for our Hoare logic. They represent a practically relevant subject and include the important pointer program features, which pose problems for verification.

4 Formalization of BDDs

4.1 State-Space

In order to represent all variables and the heap used in the program we use Isabelle records. Our model of the heap follows Burstall's [4] idea, recently emphasized by Bornat [1] and Mehta and Nipkow [8]: we have one heap f of type *ref* $\Rightarrow$ *value* for each component f of type *value* of a BDD-node. Type *ref* is our abstract view on addresses. It is isomorphic to the natural numbers and contains *Null* as an element. Figure 2 shows the C-style structure for a DAG node and the corresponding *Isabelle* records we use to represent the state-space. A C-like selection `p->var` becomes function application *var p* in our model. The global components (in our case the split heap) are grouped together in one field *globals* of the state-space-record. The remaining fields are used for local variables and procedure parameters. The semantics of our programming language model is defined via updates on this state-space [13]. The separation of global and local components is used to handle procedure calls.

Every BDD-node contains a variable *var* which is encoded as a natural number. We reserve the variables 0 and 1 for the terminal nodes. Besides every node needs pointers to its children (fields *left* and *right*) and its representative (*rep*), used during the normalization algorithm. The *next* pointer links together all nodes of the same (variable) level in the bdd. This is implemented in procedure *Levellist* as marking algorithm using the *mark* field.

```
struct node {
 nat var;
 struct node* left;
 struct node* right;
 struct node* rep;
 struct node* next;
 bool mark;
};
```

record *heap* =
  var :: *ref* ⇒ *nat*
  left :: *ref* ⇒ *ref*
  right :: *ref* ⇒ *ref*
  rep :: *ref* ⇒ *ref*
  next :: *ref* ⇒ *ref*
  mark :: *ref* ⇒ *bool*

record *state* =
  globals :: *heap*
  p :: *ref*
  levellist :: *ref list*
  . . . *local variables/parameters* . . .

Fig. 2. Node struct and program state

4.2 BDD Model

We follow the approach in [8] to abstract the pointer structure in the heap to HOL datatypes. For the formalization of BDDs we work on two levels of abstraction, the decision tree (BDT) and the graph structure (DAG). On the higher level, we describe the underlying decision tree with the datatype *bdt*:

datatype *bdt* = *Zero* | *One* | *Bdt-Node bdt nat bdt*

A *bdt* is modeled by the constructors *Zero* and *One*, which represent the terminal values *False* and *True*, and by the constructor *Bdt-Node* representing a nonterminal node with two sub-BDTs and the current decision variable.

When looking at this datatype, it becomes clear that we cannot express the concept of sharing by using this content-based definition. Therefore, we introduce another formalization level in order to describe the graph structure of the BDD based on references. For the representation of a BDD in the heap we use datatype *dag*, which is a directed acyclic graph of binary degree:

datatype *datatype dag* = *Tip* | *Node dag ref dag*

A DAG in the heap is either constant *Tip*, which is equal to the Null pointer, or a node consisting of a reference for the root node and two sub-DAGs. This representation allows us to express sharing by equal references in the nodes. Moreover it is convenient to write recursive predicates and functions on a datatype. For example *set-of* yields the references stored in the DAG:

set-of:: *dag* ⇒ *ref set*
set-of Tip = {}
set-of (*Node lt* r *rt*) = {r} ∪ *set-of lt* ∪ *set-of rt*

To actually abstract the pointer structure in the heap to the datatype *dag* we introduce the predicate *Dag*. It constructs a DAG from the initial pointer and the mappings for left and right children:

Dag:: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ *dag* ⇒ *bool*
Dag p l r Tip = (*p*=*Null*)
Dag p l r (*Node lt a rt*) = (*p*=a ∧ *p*≠*Null* ∧ *Dag* (*l p*) *l r lt* ∧ *Dag* (*r p*) *l r rt*)

This expression is true when starting at pointer *p* and following heaps *l* and *r* we can construct the DAG passed as fourth argument. The heaps *l* and *r* correspond to the fields *left* and *right* in the state-space.

To construct the decision tree out of the DAG we introduce the function *bdt*. It takes a function indicating the variables assigned to each reference as parameter. Usually we use field *var* in the state-space-record for this purpose. The result of *bdt* is an option type. This implies that not every *dag* encodes a *bdt*. The terminal nodes of the decision tree, *Zero* and *One*, are represented by an inner node *Node Tip p Tip* in the DAG, where *var p = 0* or *var p = 1*, respectively. So every proper DAG will end up in those inner nodes.

```
bdt :: dag ⇒ (ref ⇒ nat) ⇒ bdt option
bdt Tip var = None
bdt (Node Tip p Tip) var =
    (if var p = 0 then Some Zero else if var p = 1 then Some One else None)
bdt (Node Tip p (Node l p₂ r)) var = None
bdt (Node (Node l p₁ r) p Tip) var = None
bdt (Node (Node l₁ p₁ r₁) p (Node l₂ p₂ r₂)) var =
    (if var p = 0 ∨ var p = 1 then None
     else case bdt (Node l₁ p₁ r₁) var of None ⇒ None
          | Some t₁ ⇒
            case bdt (Node l₂ p₂ r₂) var of None ⇒ None
            | Some t₂ ⇒ Some (Bdt-Node t₁ (var p) t₂))
```

4.3 Properties on BDDs

We now define predicates and functions on our BDD model that we use for the specification and verification of the normalization algorithm.

Eval Function *eval* on BDTs expects the BDT which shall be evaluated and an environment (a list containing the values for all variables). It traverses the given BDT following the path indicated by the variable values and finally returns the resulting Boolean value. So *eval t* denotes the Boolean function that is represented by the decision tree *t*.

```
eval :: bdt ⇒ bool list ⇒ bool
eval Zero env = False
eval One env = True
eval (Bdt-Node l v r) env = (if env ! v then eval r env else eval l env)
```

Since all functions in HOL are total, indexing the list in *env ! v* will yield an legal but indefinite value when the index is out of range.

An interesting concept which arises from this function is eval-equivalence represented by the operator $\sim$:

$bdt_1 \sim bdt_2 \equiv eval\ bdt_1 = eval\ bdt_2$

Two BDTs are eval-equivalent if they represent the same Boolean function.

Reduced We call a DAG *reduced* if left and right non-*Tip* children differ for every node contained in it. Note that the order of equations is significant in this definition. *reduced* :: *dag* ⇒ *bool*

$reduced\ Tip = True$
$reduced\ (Node\ Tip\ p\ Tip) = True$
$reduced\ (Node\ l\ p\ r) = (l \neq r \wedge reduced\ l \wedge reduced\ r)$

Ordered The variable ordering of a given BDD is checked by predicate *ordered*. The root node stores the highest variable and the variables decrease on a path down to the leaf. For the variable information, the function takes a mapping of references to their variables (usually field *var*).

$ordered :: dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$ordered\ Tip\ var = True$
$ordered\ (Node\ (Node\ l_1\ p_1\ r_1)\ p\ (Node\ l_2\ p_2\ r_2))\ var =$
$\quad (var\ p_1 < var\ p \wedge var\ p_2 < var\ p) \wedge$
$\quad (ordered\ (Node\ l_1\ p_1\ r_1)\ var) \wedge (ordered\ (Node\ l_2\ p_2\ r_2)\ var)$
$ordered\ (Node\ Tip\ p\ Tip)\ var = True$

If the DAG properly encodes a decision tree (according to *bdt*), both children of an inner node will either be *Tip*s or again inner nodes. So we do not have to care about the other cases in the definition above.

Shared Bryant [3] calls two BDDs isomorphic if they represent the same decision tree. If a BDD is shared, then all isomorphic sub-BDDs will be represented by the same root pointer, i.e. the same DAG. This is encapsulated in predicate *isomorphic-dags-eq*, which should be read "*if two dags are isomorphic, then they are equal*":

$isomorphic\text{-}dags\text{-}eq :: dag \Rightarrow dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$isomorphic\text{-}dags\text{-}eq\ st_1\ st_2\ var \equiv$
$\forall\, bdt_1\ bdt_2.$
$\quad bdt\ st_1\ var = Some\ bdt_1 \wedge bdt\ st_2\ var = Some\ bdt_2 \wedge (bdt_1 = bdt_2) \longrightarrow$
$\quad st_1 = st_2$

i.e. if the decisions trees resulting from st_1 and st_2 are equal, the two DAGs must also be equal. The sub-DAG structure forms a partial order:

$(t < Tip) = False$
$(t < Node\ l\ p\ r) = (t = l \vee t = r \vee t < l \vee t < r)$
$s \leq t \equiv s = t \vee s < t$

In order to express that a DAG is (maximally) *shared*, we argue that all its sub-DAGs respect the *isomorphic-dags-eq* property:

$shared :: dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$shared\ t\ var \equiv \forall\, st_1\ st_2.\ st_1 \leq t \wedge st_2 \leq t \longrightarrow isomorphic\text{-}dags\text{-}eq\ st_1\ st_2\ var$

5 Normalization

BDD normalization is a central algorithm of [3] and quite complex in specification and verification. By concentrating on this part, we hope to give an impression of the provability and the verification complexity of pointer programs.

5.1 Overview of the Process of Normalization

We call the process of converting an ordered DAG into an ordered, shared and reduced DAG "normalization". It is encapsulated in procedure *Normalize* which calls on its part the sub-procedures *Levellist*, *ShareReduceRepList* and *Repoint*. The implementation follows the procedure called "reduce" in Bryant's paper [3], but imposes some simplifications and the decomposition into sub-procedures to structure the algorithm and the verification (e.g. in Bryants algorithm steps 2 and 3 below are done simultaneously).

From a high-level point of view, one can divide the normalization process into three main stages:

1. Collect the nodes of the argument DAG according to their variable in a two dimensional level-list. At index n the level-list contains all the nodes of the DAG with variable n.
2. Calculate the representative node for each node in the DAG (and store it in the *rep* field of the node) level by level and bottom up. This means that we work on the breadth of the DAG.
3. Repoint the DAG according to the representatives.

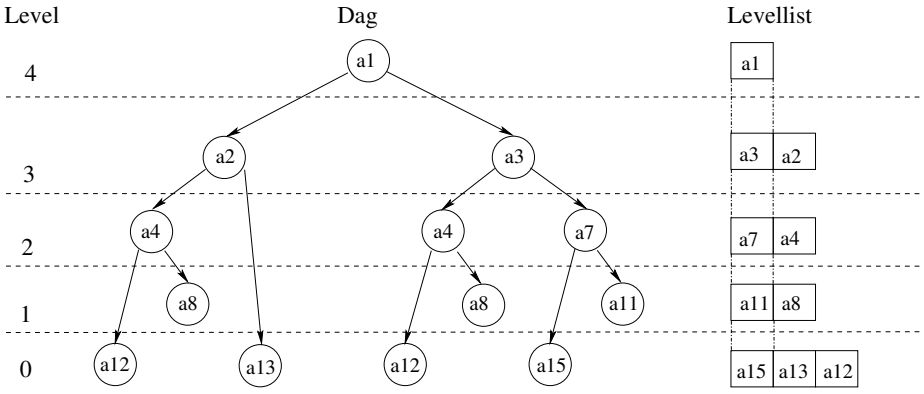


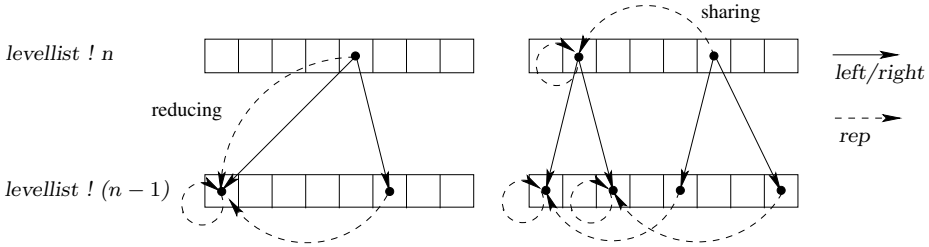
Fig. 3. Illustration of the level-list notion

We will now examine these three steps in detail:

Stage 1: Procedure *Normalize* first instantiates the level-list with an empty node list for each variable, that can be contained in the argument DAG. The necessary size of the list is given by the variable stored in the root of DAG, since the DAG is *ordered*. Afterwards, the procedure calls *Levellist*, which fills the level-list with the nodes contained in the DAG. After the call of *Levellist* a node with variable i is contained in the level-list at index i (see figure 3). Note that a node in level i not necessarily denotes the depth in the DAG, since variables do not have to appear strictly consecutive on a path through the DAG.

Via the level-list we can easily access all nodes with the same variable. Those are the ones that may have to be shared. In the DAG structure, the nodes containing the same variable can be contained in different sub-DAGs and therefore are far from each other. With the concept of a level-list, these nodes are much easier to be compared and processed. The inconvenience of this process is the complexity of the conversion from the DAG to the two dimensional list, which is visible in the length of the proof.

Stage 2: After obtaining the two dimensional level-list, we traverse each level looking for a representative for each node, which is then stored in the field *rep*. So we do not change the DAG structure at this stage but just store the pointer to the representative node in the field *rep*. Finding the representatives for all nodes in one level of the level-list is realized by procedure *ShareReduceRepList*. It is important to start the normalization with the leaf children (which have the lowest variables), because procedure *ShareReduceRepList* consults the representatives of the children in order to decide if the current node will be shared or reduced. Therefore the children representatives already have to contain the final and correct representatives. Working at a specific level we can assume that the representative DAGs of the lower levels will already be shared and reduced. If both children representatives point to the same node, we reduce, otherwise we search for a node in the current level-list with the same children representatives, which means sharing:



After procedure *Normalize* has traversed all the levels of the level-list, every node contained in the DAG has got a representative by which it will be replaced in the shared and reduced DAG to be constructed. All representative nodes derive from nodes contained in the original DAG. During the process of normalization we never need to construct new nodes.

Stage 3: The only task to complete now, is the "repointerization" of the DAG. We follow the DAG of *rep*-pointers and thereby set the *left* and *right* fields in order to obtain the desired reduced, shared and ordered DAG, which represents our BDD in the heap.

In order to summarize the functionality described above, let us look at the source code of procedure *Normalize*; the auxiliary procedures can be found in the appendix.

procedures *Normalize* $(p \mid p) =$
$\grave{1}$*evellist* := *replicate* ($\acute{p} \rightarrow \acute{v}$*ar* + 1) *Null*;
$\grave{1}$*evellist* := ***CALL*** *Levellist* ($\acute{p}$, ($\neg$ $\acute{p} \rightarrow \acute{m}$*ark*) , $\grave{1}$*evellist*);
$\acute{n}$:= *0*;

WHILE (´n < *length* ¹*levellist*) ***DO***
  CALL *ShareReduceRepList*(¹*levellist* ! ´n);
  ´n := ´n + *1*
OD;
´p := ***CALL*** *Repoint* (´p)

The bar | divides the parameters in value and result parameters. In the case of *Normalize* the input and result parameter is *p*. The arrow, like in *p*→*var*, mimics the C-style combined pointer dereferencing and field selection `p->var`. Logically it is equivalent to *var p* in our heap model.

Note that the first code line initializes the the *levellist* array with *Null*-pointers. The level-list is implemented as an array of heap-lists in our programming language, where the array size is fixed by the number of variables. Arrays are represented as HOL-lists.

5.2 Hoare Annotations

Besides the pre- and postcondition around the whole procedure body, we have inserted another Hoare triple around the while loop, starting with ***SPEC***. This inner specification characterizes the important intermediate stages of the algorithm. The precondition captures the point between step 1 and 2, and the postcondition the point between 2 and 3.

In order to be able to distinguish between the different program states we fix state variables σ for the initial procedure state and τ for the program state at the beginning of the inner Hoare triple. This state fixing is part of the assertion syntax: $\{\!|\sigma.\ldots|\!\}$ abbreviates $\{s \mid s{=}\sigma\ \ldots\}$. State components decorated with the prefix ´ refer to the current state at the position of the assertion. This helps us to speak about different stages of the program state. The logical variables τ and *ll* that are introduced by ***SPEC*** are universally quantified. We will first have a look at the fully annotated procedure before going into detail on its components.

$\forall\,\sigma$ *pret prebdt*. $\Gamma\vdash_t$
$\{\!|\sigma.$ *Dag* ´*p* ¹*left* ´*right pret* $\wedge$ *ordered pret* ´*var* $\wedge$ *bdt pret* ´*var* = *Some prebdt* $\wedge$
 $(\forall$ *no*. *no* $\in$ *set-of pret* $\longrightarrow$ ´*mark no* = ´*mark* ´*p*$)|\!\}$
¹*levellist* := *replicate* (´*p*→´*var* + *1*) *Null*;
¹*levellist* := ***CALL*** *Levellist* (´*p*, ($\neg$ ´*p*→´*mark*) , ¹*levellist*);
SPEC (τ,*ll*). $\{\!|\tau.$ *Dag* $^{\sigma}$*p* $^{\sigma}$*left* $^{\sigma}$*right pret* $\wedge$
  ordered pret $^{\sigma}$*var* $\wedge$ *bdt pret* $^{\sigma}$*var* = *Some prebdt* $\wedge$
  Levellist ¹*levellist* ´*next ll* $\wedge$
  wf-ll pret ll ´*var* $\wedge$ *length* ¹*levellist* = (´*p* → ´*var*) + *1* $\wedge$
  wf-marking pret $^{\sigma}$*mark* ´*mark* ($\neg$ $^{\sigma}$*mark* $^{\sigma}$*p*) $\wedge$
  $(\forall$ *pt*. *pt* $\notin$ *set-of pret* $\longrightarrow$ $^{\sigma}$*next pt* = ´*next pt*) $\wedge$
  ¹*left* = $^{\sigma}$*left* $\wedge$ ´*right* = $^{\sigma}$*right* $\wedge$ ´*p* = $^{\sigma}$*p* $\wedge$ ´*rep* = $^{\sigma}$*rep* $\wedge$ ´*var* = $^{\sigma}$*var*$|\!\}$
 ´*n* :=*0*;
 WHILE (´*n* < *length* ¹*levellist*)
 INV $\{\!|$*Dag* $^{\sigma}$*p* $^{\sigma}$*left* $^{\sigma}$*right pret* $\wedge$
  ordered pret $^{\sigma}$*var* $\wedge$ *bdt pret* $^{\sigma}$*var* = *Some prebdt* $\wedge$
  Levellist ¹*levellist* ´*next ll* $\wedge$

$wf\text{-}ll\ pret\ ll\ `var \wedge length\ `levellist = ((`p \rightarrow `var) + 1) \wedge$
$wf\text{-}marking\ pret\ ^{\sigma}mark\ ^{\tau}mark\ (\neg\ ^{\sigma}mark\ ^{\sigma}p) \wedge$
$^{\tau}left = ^{\sigma}left \wedge\ ^{\tau}right = ^{\sigma}right \wedge\ ^{\tau}p = ^{\sigma}p \wedge\ ^{\tau}rep = ^{\sigma}rep \wedge\ ^{\tau}var = ^{\sigma}var \wedge$
$`n \leq length\ ^{\tau}levellist \wedge$
$(\forall\, no \in Nodes\ `n\ ll.$ $\quad (*\ reduced,\ ordered\ and\ eval\ equivalent\ *)$
$no{\rightarrow}`rep{\rightarrow}`var \leq no{\rightarrow}`var \wedge$
$(\exists\, t\ rept.\ Dag\ no\ `left\ `right\ t \wedge$
$Dag\ (`rep\ no)\ (`rep \propto `left\ )\ (`rep \propto `right\ )\ rept \wedge$
$reduced\ rept \wedge ordered\ rept\ `var \wedge$
$(\exists\, nobdt\ repbdt.\ bdt\ t\ `var = Some\ nobdt \wedge$
$bdt\ rept\ `var = Some\ repbdt \wedge nobdt \sim repbdt) \wedge$
$set\text{-}of\ rept \subseteq `rep\ `\ Nodes\ `n\ ll \wedge$
$(\forall\, no \in set\text{-}of\ rept.\ `rep\ no = no))) \wedge$
$(\forall\, t_1\ t_2.$ $\quad (*\ shared\ *)$
$\{t_1,t_2\} \subseteq Dags\ (`rep\ `(Nodes\ `n\ ll))\ (`rep \propto `left\ )\ (`rep \propto `right)$
$\longrightarrow isomorphic\text{-}dags\text{-}eq\ t_1\ t_2\ `var) \wedge$
$`rep\ `\ Nodes\ `n\ ll \subseteq Nodes\ `n\ ll \wedge$
$(\forall\, pt\ i.\ pt \notin set\text{-}of\ pret \vee (`n \leq i \wedge pt \in set\ (ll\ !\ i) \wedge i < length\ `levellist)$
$\longrightarrow\ ^{\sigma}rep\ pt = `rep\ pt) \wedge$
$`levellist = ^{\tau}levellist \wedge\ `next = ^{\tau}next \wedge\ `mark = ^{\tau}mark \wedge$
$`left = ^{\sigma}left \wedge\ `right = ^{\sigma}right \wedge\ `p = ^{\sigma}p \wedge\ `var = ^{\sigma}var\}$

VAR $MEASURE\ (length\ `levellist - `n)$
DO
CALL $ShareReduceRepList(`levellist\ !\ `n);$
$`n := `n + 1$
OD
$\{(\exists\, rept.\ Dag\ (`rep\ `p)\ (`rep \propto `left\ )\ (`rep \propto `right\ )\ rept \wedge$
$reduced\ rept \wedge ordered\ rept\ `var \wedge shared\ rept\ `var \wedge$
$set\text{-}of\ rept \subseteq set\text{-}of\ pret \wedge$
$(\exists\, repbdt.\ bdt\ rept\ `var = Some\ repbdt \wedge prebdt \sim repbdt) \wedge$
$(\forall\, no \in set\text{-}of\ rept.\ (`rep\ no = no))) \wedge$
$ordered\ pret\ ^{\sigma}var \wedge\ ^{\sigma}p \neq Null \wedge$
$(\forall\, no.\ no \in set\text{-}of\ pret \longrightarrow `mark\ no = (\neg\ ^{\sigma}mark\ ^{\sigma}p)) \wedge$
$(\forall\, pt.\ pt \notin set\text{-}of\ pret \longrightarrow\ ^{\sigma}rep\ pt = `rep\ pt) \wedge$
$`levellist = ^{\tau}levellist \wedge\ `next = ^{\tau}next \wedge\ `mark = ^{\tau}mark \wedge$
$`left = ^{\sigma}left \wedge\ `right = ^{\sigma}right \wedge\ `p{=}^{\sigma}p\}$;
$`p := ***CALL*** Repoint\ (`p)$
$\{(\exists\, postt.\ Dag\ `p\ `left\ `right\ postt \wedge$
$reduced\ postt \wedge ordered\ postt\ ^{\sigma}var \wedge shared\ postt\ ^{\sigma}var \wedge$
$set\text{-}of\ postt \subseteq set\text{-}of\ pret \wedge$
$(\exists\, postbdt.\ \ bdt\ postt\ ^{\sigma}var = Some\ postbdt \wedge prebdt \sim postbdt)) \wedge$
$(\forall\, no.\ no \in set\text{-}of\ pret \longrightarrow `mark\ no = (\neg\ ^{\sigma}mark\ ^{\sigma}p)) \wedge$
$(\forall\, pt.\ pt \notin set\text{-}of\ pret \longrightarrow\ ^{\sigma}rep\ pt = `rep\ pt \wedge\ ^{\sigma}left\ pt = `left\ pt \wedge$
$^{\sigma}right\ pt = `right\ pt \wedge\ ^{\sigma}mark\ pt = `mark\ pt \wedge\ ^{\sigma}next\ pt = `next\ pt)\}$

The Precondition of procedure *Normalize* assumes all the facts that are essential for the call of its sub-procedures: The argument pointer must construct a DAG *pret*, which is ordered, and transformable into the decision tree *prebdt*.

Because of *Levellist* being a marking algorithm, all the nodes in this DAG must be identically marked.

The Postcondition The result of the procedure is a new DAG (*postt*), which is reduced, ordered, and shared. Its nodes are a subset of the nodes of the argument DAG. The decision tree *postbdt* resulting from the new DAG is "eval-equivalent" (operator $\sim$) to the decision tree that we get from the argument DAG, i.e. the Boolean function represented by the normalized BDD is still the same. Besides the marking is inverted in comparison to the beginning of the procedure (which is performed by procedure *Levellist* during the level-list construction). The rest of the postcondition states that, for nodes which are not contained in the original DAG, the fields that are normally modified by the procedure will remain unchanged. The fact that field *var* does not change, is not captured by this postcondition. We use an additional specification that exploits our split heap model and lists all the global state components that may be modified:

$\forall \sigma.\ \Gamma \vdash \{\sigma\}\ \acute{p} := \textbf{\textit{CALL}}\ \textit{Normalize}\ (\acute{p})$
$\{t.\ t\ \textit{may-only-modify-globals}\ \sigma\ \textit{in}\ [\textit{rep},\textit{mark},\textit{left},\textit{right},\textit{next}]\}$

The verification condition generator makes use of this extra specification [13]. Therefore the regular postcondition only has to mention properties of the global entities that potentially do change. That means, a procedure specification can focus on the relevant portions of the state-space.

The Inner Hoare Triple surrounds the while loop contained in the procedure. Its precondition contains the outer procedure's precondition completed by the results of the call to procedure *Levellist* and some propositions specifying the fields which remained unchanged since the beginning of the procedure. Note that we only have to mention those parts of the state-space here that we refer to in subsequent assertions, i.e. those that are relevant for our current verification task. Procedure *Levellist* adds the following assertions to our precondition:

Levellist $\grave{}$*levellist* $\acute{}$*next ll* The constructed level-list is abstracted to the two dimensional HOL-list *ll* of type *ref list list*. The array $\grave{}$*levellist* contains the initial pointers to the heap lists that link together the nodes of the same level via the $\acute{}$*next* pointers:

$\textit{Levellist levellist next ll} \equiv$
$\textit{map first ll} = \textit{levellist} \wedge (\forall i{<}\textit{length levellist}.\ \textit{List}\ (\textit{levellist}\ !\ i)\ \textit{next}\ (\textit{ll}\ !\ i))$

$\textit{first ps} \equiv \textsf{case}\ \textit{ps}\ \textsf{of}\ [] \Rightarrow \textit{Null} \mid p\ \#\ \textit{rs} \Rightarrow p$

$\textit{List p next}\ [] = (p = \textit{Null})$
$\textit{List p next}\ (a\ \#\ \textit{ps}) = (p = a \wedge p \neq \textit{Null} \wedge \textit{List}\ (\textit{next p})\ \textit{next ps})$

wf-ll pret ll $\acute{}$*var* The level-list is well-formed, i.e. all nodes in the argument DAG are contained in the level-list on their variable position and all nodes in the level-list are contained in the argument DAG:

$\textit{wf-ll pret ll var} \equiv$
$(\forall p.\ p \in \textit{set-of pret} \longrightarrow p \in \textit{set}\ (\textit{ll}\ !\ \textit{var}\ p)) \wedge$
$(\forall k{<}\textit{length ll}.\ \forall p{\in}\textit{set}\ (\textit{ll}\ !\ k).\ p \in \textit{set-of pret} \wedge \textit{var}\ p = k)$

length ‵*levellist* = (‵*p*→‵*var*) + *1* The length of the level-list fits to the variables contained in the DAG.

wf-marking pret $^{\sigma}$*mark* ‵*mark* (¬ $^{\sigma}$*mark* $^{\sigma}$*p*) All nodes in the DAG are marked contrary to their initial marking:

wf-marking pret mark-old mark-new marked ≡
case *pret* **of** *Tip* ⇒ *mark-new* = *mark-old*
| *Node lt p rt* ⇒
(∀ *n*. *n* ∉ *set-of pret* ⟶ *mark-new n* = *mark-old n*) ∧
(∀ *n*. *n* ∈ *set-of pret* ⟶ *mark-new n* = *marked*)

Now let us think about the inner postcondition. The only action taken after the inner Hoare triple in the source code is the call to procedure *Repoint*. *Repoint* only redirects the original DAG *pret* to the DAG of representatives *rept*, which already has the desired properties: it is reduced, ordered, shared and the resulting decision tree *repbdt* encodes the same Boolean function as the original one. Moreover, since every node in *rept* is a representative the *rep* field of those nodes will point to the node itself. The DAG of representatives *rept* can be obtained out of the original DAG by following the *rep* pointers: *Dag* (‵*rep* ‵*p*) (‵*rep* ∝ ‵*left*) (‵*rep* ∝ ‵*right*) *rept*. We begin with the representative of the root pointer ‵*rep* ‵*p*, and instead of just following the *left* and *right* pointers we make the additional indirection through *rep*, by the infix operator ∝. It is defined as an extension of function composition avoiding to consider representatives of a *Null* pointer:

rep ∝ *f* ≡ λ*p*. **if** *f p* = *Null* **then** *Null* **else** (*rep* ∘ *f*) *p*

So in case *left p* ≠ *Null*, the expression (*rep* ∝ *left*) *p* is equivalent to two dereferences: *p*→*left*→*rep*.

In addition we preserve some facts that we already know, like the inversion of the marks, and add the assertion about the parts of the state that are not modified in the loop. Note that we do not modify the DAG structure, since the fields *left* and *right* remain unchanged. Only the *rep* field is modified.

The Loop Invariant starts with the repetition of the facts that we already know from the precondition of the inner Hoare triple. After that the main part of the invariant describes the properties of the processed levels that we have to lift to the current level while proving the invariant and that must suffice to derive the postcondition (of the inner triple) after the loop. Intuitively we have to express that all sub-BDDs stemming from the representative nodes are ordered, reduced and shared and encode the same Boolean function as their original counterparts.

To get hold of the processed nodes and DAGs, we introduce two more predicates, which express that we are processing the original DAG level by level:

- *Nodes i ll* ≡ $\bigcup_{k \in \{k \mid k < i\}}$ *set* (*ll* ! *k*)
  Nodes helps us to speak of all nodes, which are contained in the DAG or level-list up to level i.
- *Dags nodes left right* ≡ {*t* | ∃ *p*. *Dag p left right t* ∧ *t* ≠ *Tip* ∧ *set-of t* ⊆ *nodes*}
  A DAG is contained in *Dags nodes left right* if its nodes are all contained in

nodes, if it forms a DAG based on the fields *left* and *right* and if this DAG is no Tip.

For every node *no* that is already processed we know that the representative will not point to a bigger variable; during sharing the variable remains the same and reducing decreases the variable. Starting from a node *no* we can construct the DAG and the decision tree following the original pointers (t and *nobdt*) and following the representative pointers (*rept* and *repbdt*). The representative DAG *rept* is ordered and reduced, and the encoded Boolean function is preserved ($nobdt \sim repbdt$). The nodes of *rept* all are representatives of nodes we have processed so far. This is expressed by $set\text{-}of\ rept \subseteq \acute{r}ep\ `\ Nodes\ \acute{n}\ ll$. Here the infix ` is the set image operation. So we can rephrase the set on the right hand side with $\{\acute{r}ep\ no \mid no \in Nodes\ \acute{n}\ ll\}$. Moreover the representative of an representative node will point to the node itself. This ensures uniqueness of the representatives.

To properly express the sharing of the representative DAGs, we cannot only refer to a single DAG constructed from a representative node, since we also have to consider sharing between all sub-BDDs. For every two DAGs t_1 and t_2 that we construct from the representative nodes, the sharing property *isomorphic-dags-eq* has to hold.

The remaining parts of the invariant express that the representative nodes are contained in the original nodes, and describe the parts of state that remain unmodified by the loop.

The Loop Variant justifies termination and is specified via a wellfounded relation. In this case a measure function, expressing that the distance of the loop variable to the length of the level-list decreases.

Both Verma et al. [14] as well as Kristic and Matthews [7] encounter some problems regarding termination. They directly map their BDD-algorithms to recursive functions in Coq or Isabelle/HOL respectively. Since the underlying logics only support total functions, they have to come up with a justification for termination upon function definition. The recursive algorithms on BDDs only terminate for proper inputs (e.g. no cycles). Verma et. al. work around this problem by formally defining the recursion on an artificial counter (the variable level). Kristic and Matthews come up with a scheme to simultaneously define the function together with an invariant. By this they are able to handle the nested recursion, that occurs because the global state is an explicit parameter of their functions. Subsequent function application on the left and right sub-DAG results in nested recursion in their approach.

These problems do not occur in our model (e.g. for the auxiliary procedures *Levellist* or *Repoint*), since we do not directly define them as functions in HOL, but just define the piece of syntax making up the procedure body. We can easily restrict the input to well-formed BDDs by the precondition of the Hoare triple, e.g. $Dag\ \acute{p}\ \grave{l}ow\ \grave{h}igh\ pret$ already ensures that there are no cycles in the pointer structure.

6 Conclusion

The verification of partial correctness of the normalization algorithm and its auxiliary procedures sums up to about 10000 lines of Isabelle/Isar formalization and proofs and is based on a master thesis [11]. Adapting the proofs to total correctness is straightforward and only adds a few lines.

We locate the reasons of the complexity mainly in the data structure, which involves a high degree of data sharing and side effects, which results in quite complex invariants, specifications and proofs. We have to keep track of the original BDD the level-list and the representative BDD. As an example our proof that the property marked as $(* \textit{ shared } *)$ in the invariant is preserved, while we proceed from level n to $n + 1$, required about 1000 lines. We consider two arbitrary Dags up to level $n + 1$ and have to show the *isomorphic-dags-eq* property for them. We make a case distinction, whether both Dags are already in level n, one Dag is already in level n, or none of them is in level n. In the latter case we proceed by inspecting the root nodes to decide whether they where shared or not. Those kind of case distinctions for various properties add up to the large proofs.

To prove the verification conditions, we used the structured language Isar [9] that allows to focus on and keep track of the various aspects of the proof, so that we can conduct it in a sensible order. Moreover it turned out that the Isar proofs are quite robust with regard to the iterative adaption of the invariant resulting from failed proof attempts. The already established lines of reasoning remained stable, while adding new aspects to, or strengthening parts of the invariant. The relatively large size of the proofs is partly explained by the fact that the declarative Isar proofs are in general more verbose than tactic scripts.

The Hoare logic framework and the split heap model appeared to form a suitable verification environment on top of Isabelle/HOL. The abstraction of pointer structures to HOL datatypes allows us to give reasonable specifications. The split heap model addresses parts of the separation problems that occur when specifying procedures on pointer structures. The overhead of describing the parts of the heap that do not change is kept small. The main effort of the work goes into the problem and not into the framework.

The model we used to describe the properties of the BDD pointer structure can serve as a solid basis for more involved BDD algorithms.

References

1. Richard Bornat. Proving pointer programs in Hoare logic. In J. Oliveira R. Backhouse, editor, *Mathematics of Program Construction*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer, 2000.
2. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE*, pages 40–45, june 1990.
3. Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

4. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
5. Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mie Barnett, editors, *Formal Methods and Software Engineering 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004*, volume 3308 of *Lect. Notes in Comp. Sci.* Springer, 2004.
7. Sava Krstic and John Matthews. Verifying BDD algorithms through monadic interpretation. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation: Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22*, volume 2294 of *LNCS*, pages 182–195, 2002.
8. Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *Lect. Notes in Comp. Sci.*, pages 121–135. Springer, 2003.
9. Tobias Nipkow. Structured Proofs in Isar/HOL. In *TYPES 2002*, volume 2646 of *Lect. Notes in Comp. Sci.* Springer, 2002. http://isabelle.in.tum.de/docs.html.
10. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
11. Veronika Ortner. Verification of BDD Algorithms. Master's thesis, Technische Universität München, 2004. available from http://www.veronika.langlotz.info/.
12. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.
13. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lect. Notes in Art. Int.*, pages 398–414. Springer-Verlag, 2005.
14. Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, volume 1961, pages 162–181. Springer, 2000.

A Auxiliary Procedures

Levellist traverses the DAG, puts unmarked nodes to the front of the corresponding level-list slot, and switches their mark. Marking ensures that nodes are only collected once and thus no cycles are introduced in the list.

procedures *Levellist* (p, m, *levellist* | *levellist*) =
IF ($p \neq Null$) ***THEN***
  IF ($p\rightarrow mark \neq m$) ***THEN***
    $levellist$:= ***CALL*** *Levellist* ($p\rightarrow left, m, levellist$);
    $levellist$:= ***CALL*** *Levellist* ($p\rightarrow right, m, levellist$);

```
      ´p→´next := `levellist ! (´p→´var);
      `levellist ! (´p→´var) := ´p;
      ´p→´mark := ´m;
   FI
FI
```

ShareReduceRepList processes one level of the level-list. Non-leaf nodes with the same children representatives are reduced, all the others are shared.

isLeaf-pt p left right ≡ *left p* = *Null* ∧ *right p* = *Null*

```
procedures ShareReduceRepList (nodeslist) =
´node := ´nodeslist;
WHILE (´node ≠ Null) DO
  IF (¬ isLeaf-pt ´node `low `high ∧ ´node→`low→´rep = ´node→`high→´rep)
  THEN ´node → ´rep := ´node → `low → ´rep  (* reducing *)
  ELSE CALL ShareRep (´nodeslist , ´node )    (* sharing *)
  FI;
  ´node := ´node→´next
OD
```

ShareRep shares node *p* by searching its representative in the current *nodeslist*. In case of leafs, the representative is the first element in the list. Otherwise the representative is the first node in the list with the same children representatives. Since *p* itself is in the list we will always find a node.

repNodes-eq p q left right rep ≡
(*rep* ∝ *right*) *p* = (*rep* ∝ *right*) *q* ∧ (*rep* ∝ *left*) *p* = (*rep* ∝ *left*) *q*

```
procedures ShareRep (nodeslist, p) =
IF (isLeaf-pt ´p `low `high)
THEN ´p → ´rep := ´nodeslist
ELSE
  WHILE (´nodeslist ≠ Null) DO
    IF (repNodes-eq ´nodeslist ´p `low `high ´rep)
    THEN ´p→´rep := ´nodeslist; ´nodeslist := Null
    ELSE ´nodeslist := ´nodeslist→´next
    FI
  OD
FI
```

Repoint traverses the DAG while re-pointing the nodes to their representatives.

```
procedures Repoint (p | p) =
IF (´p ≠ Null) THEN
  ´p := ´p→´rep;
  IF (´p ≠ Null) THEN
    ´p→`left := CALL Repoint (´p→`left);
    ´p→´right := CALL Repoint (´p→´right)
  FI
FI
```