BDDs verified in a proof assistant
(Preliminary report)

Mathieu Giorgino
Martin Strecker
IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse

Abstract

This paper is a case study in mechanical verification of graph manipulating algorithms: We prove the correctness of a family of algorithms constructing Binary Decision Diagrams in a monadic style. It distinguishes itself from previous verification efforts in two respects: Firstly, building the BDD structure is guided by a primitive recursive descent which makes the proof of termination trivial. Secondly, the development is modular: it is parametrized by primitives manipulating high-level hash tables that can be realized by several implementations.

1 Introduction

Binary Decision Diagrams (BDDs) [3] are a compact way of representing Boolean formulas. They are widely used in applications such as model checking and digital circuit development. The main idea of BDDs is to represent a Boolean formula as a decision tree, to join common subtrees and eliminate certain redundancies, so as to arrive at a canonical representation of formulas. This makes it particularly easy to check for validity of a formula or equivalence of two formulas.

This paper describes, primarily, an extended case study, intended to illustrate a particular method of developing and verifying pointer algorithms. There have been previous efforts of verifying BDD algorithms, see Section 2 for a discussion. Surprisingly, they all work on a relatively low-level representation that does not exploit the tree structure inherent in BDDs. This enormously complicates some proofs, such as termination. We remain entirely on the level of trees, but in order to take sharing of subtrees into account, we adorn the nodes with references incorporating the "object identity" of the trees. Our main data structures are described in Section 3. In the course of building up a BDD, new references have to be generated, and subtrees have to be retrieved from or stored in a sort of hash table. This management of a program state is carried out "behind the scenes", in a monadic framework, which is described further in

Section 4. Nevertheless, most of our function definitions are by structural recursion, which makes the termination argument trivial and furthermore permits simple proof methods such as structural induction to be applied.

Verified BDD algorithms are of interest in their own right, for example in the context of verified decision procedures for certain modal [4] or temporal logics [12, 13]. We do not pretend to present the ultimate implementation of BDDs in this paper, but we have attempted to keep the development modular, permitting further optimizations to be integrated with little effort. We thus give an abstract specification of the functions manipulating hash tables (Section 5) and provide two distinct implementations (Section 6).

The formalization described in this paper has been carried out in the Isabelle proof assistant [10]. It uses some specificities of Isabelle, most of which are not essential (the syntax definition facilities for writing monadic programs in a readable style) or can be replaced by related concepts available in other proof assistants (the structuring mechanism of locales). The development is available on the authors' home pages [5].

2 Related Work

Most of the formalizations we are aware of follow standard expositions of BDD algorithms [3, 1] and thus do not differ substantially from an algorithmic viewpoint. The most essential differences concern the representation of the state space.

The present paper owes much to [7], which introduces the approach of verifying imperative programs in a proof assistant by representing them in a monadic style. The state space of the program is represented as a set of interconnected nodes that have to satisfy some well-formedness constraints. A major problem is the termination proof of the function *app* (see Section 5) that has to be carried out in parallel with the correctness argument: The function makes two consecutive recursive calls that, by transforming the global state space, could possibly invalidate the well-formedness conditions on which termination relies.

The formalization [14], carried out in the Coq proof assistant, is based on a similar BDD representation, but the algorithm directly accesses the program state, represented as a (nested) map. There is no attempt to hide manipulation of the state behind abstract state transformers. The above-cited termination problem is circumvented by recursing not over the structure of the BDD, but over a natural number representing an upper bound of the size of the BDD. Thus, the algorithm employs an artifact whose sole purpose it is to facilitate the representation in a proof assistant. This formalization is the most comprehensive one that we are aware of. It contains several essential optimizations (handling negation; garbage collection) that have not yet been addressed in our work.

A formalization in PVS [15] uses a tricky encoding of hash tables by injective pairing functions and can thus avoid having to handle a program state altogether – the BDD construction is entirely functional. It is not clear how this approach scales to hash tables containing a great number of elements.

The above formalizations adopt a functional representation (possibly hidden behind a monadic framework) of the BDD algorithms. A radical departure is the direct coding in an imperative language [11] in the style of C and the verification by means of a Hoare calculus. The algorithm uses an optimized representation of hash tables ("level lists"), but the full proof of correctness is complex and extends over several hundred pages.

Recognizing the huge effort to be spent on verifying imperative programs manipulating low-level data structures, we aim at providing reasoning support for an intermediate level that benefits from some performance gains of imperative wrt. functional programming (destructive updates, pointer manipulation) without abandoning high-level data structures. A companion paper [6] explores the applicability of our approach to the Schorr-Waite graph marking algorithm. Clearly, sophisticated optimizations based on bit-level manipulations are not immediately within the reach of our techniques, but could be achieved by successive data structure refinements.

3 Binary Decision Diagrams

BDDs are used to represent and manipulate efficiently Boolean expressions. We will use them as starting point of our algorithms, by defining a function constructing BDDs from them. The definition of expressions is standard:

```
datatype bbinop = OR | AND | IMP | IFF

datatype 'v expr =
  Var 'v
| Const bool
| BExpr bbinop ('v expr) ('v expr)
```

and their interpretation is done by *interp-expr* (where, obviously, *interp-bbinop* maps constructors of *bbinop* to Boolean functions):

```
primrec interp-expr :: 'v expr ⇒ ('v ⇒ bool) ⇒ bool where
  interp-expr (Var v) tab = tab v
| interp-expr (Const b) tab = b
| interp-expr (BExpr bop e1 e2) tab =
  (interp-bbinop bop) (interp-expr e1 tab) (interp-expr e2 tab)
```

In this function and other functions of interpretation, variable values are represented by a function from variables indices to Booleans. We now define BDDs as binary trees in which references are added to nodes and leaves (*rtree*):

```
datatype ('a, 'b) tree =
  Leaf 'a
| Node 'b (('a, 'b) tree) (('a, 'b) tree)

types ('r, 'a, 'b) rtree = ('a * 'r, 'b * 'r) tree
```

In this way, as long as subtrees having identic references are the same, we can represent sharing. To ensure this property giving meaning to references, we use the predicate *ref-unique ts*:

definition *ref-unique* :: (*'r*, *'a*, *'v*) *rtree set* ⇒ *bool* **where**
ref-unique ts ≡
∀ *t1 t2*. *t1* ∈ *ts* ⟶ *t2* ∈ *ts* ⟶ *ref-equal* (*t1*, *t2*) = *struct-equal* (*t1*, *t2*)

in which *ref-equal* means that two trees have the same reference attribute, and *struct-equal* is structural equivalence neglecting references, thus corresponding to the typical notion of equality of data in functional languages.

While the left-to-right implication of this equivalence is the required property (two nodes having the same reference are the same), the other implication ensures the maximal sharing (same subtrees are shared, *i. e.* have the same reference).

Let us illustrate the concept of subtree sharing by an example. A non-shared BDD (thus, in fact, just a decision tree) representing the formula $(x \wedge y) \vee z$ is given by the following tree (omitting references):

```
Node x
     (Node z (Leaf false) (Leaf true)),
     (Node y (Node z (Leaf false) (Leaf true))
             (Leaf true))
```

There is a common subtree `(Node z (Leaf false) (Leaf true))` which we would like to share. We therefore adorn the tree nodes with references, using the same reference for structurally equal trees, for example:

```
Node (x, 1)
     (Node (z, 3) (Leaf (false, 4)) (Leaf (true, 5))),
     (Node (y, 2) (Node (z, 3) (Leaf (false, 4)) (Leaf (true, 5))),
                (Leaf (true, 5)))
```

The process of sharing is illustrated in Figure 1.

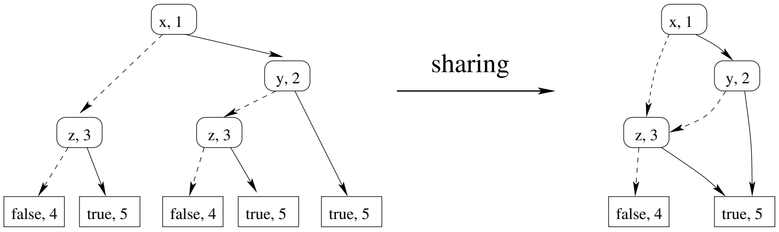

Figure 1: Sharing nodes in a tree

Each node contains a variable index whose type is any type equipped with a linear order (as indicated by Isabelle's type class annotation) and each leaf contains a value of any type instantiated later in the development (for interpretations) to Booleans. To allow writing simple and generic algorithms (*i. e.* avoid particular cases), leaves and nodes should be usable in the same way. For example, we define a linear order on levels of trees by having level of leaves always greater than levels of nodes and using variable indices to compare nodes.

BDDs can be interpreted by giving values to variables which is what the *interp* function does:

```
fun interp :: ('r, 'a, 'v) rtree ⇒ ('v ⇒ bool) ⇒ 'a where
  interp (Leaf (b,r)) - = b
| interp (Node (v,r) l h) tab = (if tab v then interp h tab else interp l tab)
```

With this definition, and without any other property, BDDs would be rather hard to manipulate. On the one hand, same variable indices could appear several times on paths from root to leaves. On the other hand, variables would not be in the same order, making comparison of BDDs harder. Moreover, a lot of space would be wasted. To circumvent this problem, one often imposes a strict order on variables, the resulting BDDs being called ordered (OBDDs):

```
fun tree-vars :: ('a, 'v * 'r) tree ⇒ 'v set where
  tree-vars (Node (v,r) l h) = insert v (tree-vars l ∪ tree-vars h)
| tree-vars (Leaf b) = {}

fun ordered where
  ordered (Leaf b) = True
| ordered (Node (i, r) l h) =
  ((∀ j ∈ (tree-vars l ∪ tree-vars h). i < j) ∧ ordered l ∧ ordered h)
```

An additional important property is to avoid redundant tests, which occurs when the two children of a node have the same interpretation. All the nodes satisfying this property can be removed. In this case, the OBDD is said to be reduced (ROBDD).

```
fun reduced :: ('r, 'a, 'v) rtree ⇒ bool where
  reduced (Node vr l h) = ((interp l ≠ interp h) ∧ reduced l ∧ reduced h)
| reduced (Leaf -) = True
```

This property uses a high-level definition (*interp*), but it can be deduced (*c.f.* lemma *non-redundant-imp-reduced* below) from the three low-level properties *ref-unique*, *ordered* (already seen) and *non-redundant*:

```
fun non-redundant :: ('r, 'a, 'v) rtree ⇒ bool where
  non-redundant (Node vr l h) =
   (¬ ref-equal(l, h) ∧ non-redundant l ∧ non-redundant h)
| non-redundant (Leaf -) = True
```

We then merge these properties in two definitions *robdd* (high-level) and *robdd-refs* (low-level):

```
definition robdd t ≡ (ordered t ∧ reduced t)
definition robdd-refs t ≡ (ordered t ∧ non-redundant t ∧ ref-unique (treeset t))
```

And we can then show that ROBDDs are a canonical representation of Boolean expressions, *i. e.* that two equivalent ROBDDs are structurally equal.

lemma *canonic-robdd*:
fixes *t1* :: (-, -::*linorder* × -) *tree*
shows ⟦*robdd t1*; *robdd t2*; *interp t1* = *interp t2*⟧ ⟹ *struct-equal* (*t1*, *t2*)
lemma *non-redundant-imp-reduced*:
fixes *t* :: (-, -::*linorder* × -) *tree*
shows ⟦*ordered t*; *ref-unique* (*treeset t*); *non-redundant t*⟧ ⟹ *reduced t*
lemma *canonic-robdd-refs*:
fixes *t1* :: (-, -::*linorder* × -) *tree*
shows ⟦*robdd-refs t1*; *robdd-refs t2*; *interp t1* = *interp t2*⟧ ⟹ *struct-equal* (*t1*, *t2*)
lemma *non-redundant-reduced*:
fixes *t* :: (-, -::*linorder* × -) *tree*
shows ⟦*ref-unique* (*treeset t*); *ordered t*⟧ ⟹ *non-redundant t* = *reduced t*
lemma *robdd-refs-robdd*:
fixes *t* :: (-, -::*linorder* × -) *tree*
shows *ref-unique* (*treeset t*) ⟹ *robdd-refs t* = *robdd t*

4 Imperative Programs : Monads

This section presents a way to manipulate low-level programs. We use a heap-transformer monad providing means to reason about monadic/imperative code along with a nice syntax, and that should allow to generate similar executable code.

We first define the state-reader and state-transformer monads and a syntax seamlessly mixing them. We encapsulate them in the *SR* – respectively *ST* – datatypes, as functions from a state to a return value – respectively a pair of return value and state.

We can escape from these datatypes with the *runSR* – respectively *runST* and *evalST* – functions which are intended to be used only in logical parts (theorems and proofs) and that should not be extractible.

datatype (*'a*, *'s*) *SR* = *SR 's* ⇒ *'a*
datatype (*'a*, *'s*) *ST* = *ST 's* ⇒ *'a* × *'s*

consts
runSR :: (*'a*, *'s*) *SR* ⇒ *'s* ⇒ *'a*
runST :: (*'a*, *'s*) *ST* ⇒ *'s* ⇒ *'a* × *'s*

primrec *runSR* (*SR m*) = *m*
primrec *runST* (*ST m*) = *m*

abbreviation *evalST* :: (*'a*, *'s*) *ST* ⇒ *'s* ⇒ *'a*
where *evalST fm s* ≡ *fst* (*runST fm s*)

abbreviation *stateST* :: (*'a*, *'s*) *ST* ⇒ *'s* ⇒ *'s*
where *stateST fm s* ≡ *snd* (*runST fm s*)

The *return* (also called *unit*) and *bind* functions for manipulating the monads are then defined classicaly with the infix notations $\trianglerighteq_{SR}$ and $\trianglerighteq_{ST}$ for *bind*s.

We add also the function *SRtoST* translating state-reader monads to state-transformer monads and the function *thenST* (with infix notation $\triangleright_{ST}$) abbreviating binding without value transfer.

consts
$\mathit{returnSR} :: {'a} \Rightarrow ({'a},\ {'s})\ \mathit{SR}$
$\mathit{returnST} :: {'a} \Rightarrow ({'a},\ {'s})\ \mathit{ST}$
$\mathit{bindSR}\ \ :: ({'a},\ {'s})\ \mathit{SR} \Rightarrow ({'a} \Rightarrow ({'b},\ {'s})\ \mathit{SR}) \Rightarrow ({'b},\ {'s})\ \mathit{SR}$ (**infixr** $\trianglerighteq_{SR}$)
$\mathit{bindST}\ \ :: ({'a},\ {'s})\ \mathit{ST} \Rightarrow ({'a} \Rightarrow ({'b},\ {'s})\ \mathit{ST}) \Rightarrow ({'b},\ {'s})\ \mathit{ST}$ (**infixr** $\trianglerighteq_{ST}$)
$\mathit{SRtoST}\ \ :: ({'a},{'s})\ \mathit{SR} \Rightarrow ({'a},\ {'s})\ \mathit{ST}$

defs
$\mathit{returnSR}\ a \equiv \mathit{SR}\ (\lambda\ s.\ a)$
$\mathit{returnST}\ a \equiv \mathit{ST}\ (\lambda\ s.\ (a,\ s))$
$\mathit{bindSR}\ m\ f \equiv \mathit{SR}\ (\lambda\ s.\ (\lambda\ x.\qquad \mathit{runSR}\ (f\ x)\ s)\ \ (\mathit{runSR}\ m\ s))$
$\mathit{bindST}\ m\ f \equiv \mathit{ST}\ (\lambda\ s.\ (\lambda\ (x,\ s').\ \mathit{runST}\ (f\ x)\ s')\ (\mathit{runST}\ m\ s))$
$\mathit{SRtoST}\ \mathit{sr}\ \equiv \mathit{ST}\ (\lambda\ s.\ (\mathit{runSR}\ \mathit{sr}\ s,\ s))$

abbreviation
$\mathit{thenST} :: ({'a},\ {'s})\ \mathit{ST} \Rightarrow ({'b},\ {'s})\ \mathit{ST} \Rightarrow ({'b},\ {'s})\ \mathit{ST}$ (**infixr** $\triangleright_{ST}$ *55*)
where $a \triangleright_{ST} b \equiv a \trianglerighteq_{ST} (\lambda\ \text{-}.\ b)$

We can then verify the monad laws:

lemma *monadSRlaws* :
$\forall\ \ v\ f.\ (\mathit{returnSR}\ v) \trianglerighteq_{SR} f\ = f\ v$
$\forall\ \ a.\ a \trianglerighteq_{SR} \mathit{returnSR} = a$
$\forall\ \ (x{::}({'s},\ {'a})\ \mathit{SR})\ f\ g.\ (x \trianglerighteq_{SR} f) \trianglerighteq_{SR} g = x \trianglerighteq_{SR} (\lambda\ v.\ ((f\ v) \trianglerighteq_{SR} g))$
by(*simp-all add*: *expand-SR-eq SR-run0*)

lemma *monadSTlaws* :
$\forall\ \ v\ f.\ (\mathit{returnST}\ v) \trianglerighteq_{ST} f\ = f\ v$
$\forall\ \ a.\ a \trianglerighteq_{ST} \mathit{returnST} = a$
$\forall\ \ (x{::}({'a},\ {'s})\ \mathit{ST})\ f\ g.\ (x \trianglerighteq_{ST} f) \trianglerighteq_{ST} g = x \trianglerighteq_{ST} (\lambda\ v.\ ((f\ v) \trianglerighteq_{ST} g))$
by (*simp-all add*: *expand-ST-eq ST-run0 split*:*prod.splits*)

We define also syntax translations to use the Haskell-like *do*-notation.

The principal difference between the Haskell *do*-notation and this one is the use of state-readers for which order does not matter. With some syntax transformations, we can simply compose several state readers into one as well as give them as arguments to state writers, almost as it is done in imperative languages (for which state is the heap). In an adapted context – *i. e.* in $\mathit{doSR}\{\dots\}$ or $\mathit{doST}\{\dots\}$ – we can so use state readers in place of expressions by simply putting them in $\langle\dots\rangle$, the current state being automatically provided to them, only thanks to the syntax transformation which propagates the same state to all $\langle\dots\rangle$.

For example with $f^{'a \Rightarrow ('b,\ 's)\ ST}$, $a^{('a,'s)\ SR}$, $g^{((),\ 's)\ ST}$ and $h^{'b \Rightarrow\ 'd}$, all these expressions are equivalent:

- $\mathit{doST}\ \{\ x \leftarrow f\ \langle a \rangle;\ g;\ \mathit{returnST}\ (h\ x)\}$
- $\mathit{doST}\ \{\ va \leftarrow \mathit{SRtoST}\ a;\ x \leftarrow f\ va;\ g;\ \mathit{returnST}\ (h\ x)\}$
- $\mathit{doST}\ \{\ x \leftarrow \mathit{ST}\ (\lambda s.\ \mathit{runST}\ (f\ (\mathit{runSR}\ a\ s))\ s);\ g;\ \mathit{returnST}\ (h\ x)\}$
- $\mathit{ST}\ (\lambda s.\ \mathit{runST}\ (f\ (\mathit{runSR}\ a\ s))\ s) \trianglerighteq (\lambda x.\ g \triangleright \mathit{returnST}\ (h\ x))$

The notions introduced so far are appropriate for manipulating an existing set of references. We now define a type class allowing infinite generation of values, which will be useful for allocating new references:

class $\mathit{genr} = \mathit{eq}\ +$
  fixes $\mathit{gen} ::\ {}'a\ \mathit{list} \Rightarrow {}'a$
  assumes $\mathit{gen\text{-}def}$ $[\mathit{rule\text{-}format},\ \mathit{simp}]$:
    $\forall\ \mathit{vs}.\ (\mathit{gen}\ \mathit{vs}) \notin \mathit{set}\ \mathit{vs}$

We then define the heap we will use as the state in the state-reader/transformer monads. We represent references with a very simple datatype, only used as a tag:

datatype $'n\ \mathit{ref} = \mathit{Ref}\ 'n$

We represent the heap by an extensible record containing a field *val* being an association list of references and objects in the heap. This choice allows to have a heap of finite size, making allocation to always be possible without restricting the state.

record $('n,\ 'v)\ \mathit{heap} =$
  $\mathit{val} :: ('n\ \mathit{ref} \times 'v)\ \mathit{list}$

We then also define some abbreviations for simplifying access to the heap:

abbreviation
  $\mathit{refs}\ s \equiv \mathit{map}\ \mathit{fst}\ (\mathit{val}\ s)$

abbreviation
  $\mathit{heap}\ h\ n \equiv (\mathit{case}\ \mathit{map\text{-}of}\ (\mathit{val}\ h)\ n\ \mathit{of}\ \mathit{Some}\ v \Rightarrow v)$

and we define primitives to read and write the heap:

consts
  $\mathit{read}\ \ ::\ 'n\ \mathit{ref} \Rightarrow ('v,\ ('n,\ 'v,\ 'a)\ \mathit{heap\text{-}scheme})\ \mathit{SR}$
  $\mathit{write}\ ::\ ['n\ \mathit{ref},\ 'v] \Rightarrow (\mathit{unit},\ ('n,\ 'v,\ 'a)\ \mathit{heap\text{-}scheme})\ \mathit{ST}$ (**infix** $.\!=\ 15$)
  $\mathit{alloc}\ ::\ ('n\ \mathit{ref} \Rightarrow 'v) \Rightarrow ('n\ \mathit{ref},\ ('n{::}\mathit{genr},\ 'v,\ 'a)\ \mathit{heap\text{-}scheme})\ \mathit{ST}$
  $\mathit{new}\ \ ::\ 'v \Rightarrow ('n\ \mathit{ref},\ ('n{::}\mathit{genr},\ 'v,\ 'a)\ \mathit{heap\text{-}scheme})\ \mathit{ST}$

5 Constructing BDDs

5.1 Main steps of the construction

Our BDD construction algorithm is inspired by the presentation in [1]. In the following, we give a high-level summary of the main construction steps, before discussing the functions in detail further below:

1. We recall that the purpose of BDD construction is to convert an expression (of type *expr*) to a ROBDD, which is a canonical representation of this expression. This is accomplished by function *build* which traverses the expression, recursively builds up BDDs of the subexpressions and, depending on the Boolean function represented by the outermost constructor, combines these with the aid of a function *app*.

2. *app* takes a Boolean function and two BDDs and traverses them in parallel until reaching the leaf positions, where the Boolean function is applied to the leaf values. During recursive descent, two BDDs l and h are constructed, one for the “low” and one for the “high” branch, and are then combined (function *mk*) to form the root of the new BDD. Using memoization techniques, it is in some cases possible to avoid a descent down to the leaves. This has not been implemented here, but would not be a major difficulty.

3. *mk* takes a variable index i (determined according to a previously fixed variable order) and two BDDs l and h. If a BDD with root i and sub-BDDs l and h already exists, *mk* returns it, otherwise it constructs a new BDD.

It is at this point that we need to access a hash table associating triples (i, l, h) to BDDs, and of course, this table is modified by our functions and therefore has to be passed on to subsequent operations. This motivates the monadic style of our functions, whose definitions are presented in Section 5.3.

5.2 Abstracting from hash tables

The precise structure of the hash table is immaterial for the BDD algorithm itself, as long as we know how to interact with it. We will now give a specification based on Isabelle's locale mechanism [2], and provide two implementations in Section 6 further below. A similar structuring principle is employed in the Isabelle Collection Framework [8].

In Section 4, we have described the monadic background theory that introduces the notion of *heap* and provides support for handling references. We now extend the state space with components *trees* (the set of trees stored in the hash table) and *constTrue* and *constFalse*, representing the pre-allocated leaf nodes for *true* and *false*;

```
record ('a, 'r, 'v) bdd-state = ('r, unit) heap +
  trees :: ('a * 'r ref, 'v * 'r ref) tree set
  constTrue :: ('a * 'r ref, 'v * 'r ref) tree
  constFalse :: ('a * 'r ref, 'v * 'r ref) tree
```

The locale defines the functions

- *add i l h* for allocating a new BDD node with variable index i and sub-BDDs l and h.

- *lookup i l h* that returns *Some n*, if *n* is a node stored in the table having variable *i* and sub-BDDs *l* and *h*. If no such node exists, the function returns *None*.

- *constLeaf b* returns the *constTrue* resp. *constFalse* leaf.

Additionally, the locale definition contains a morphism *to_bdd_state* mapping the representation *'s* to its abstraction *('a, 'r, 'v) bdd_state*, and a representation invariant *invar* whose purpose will become clear once we describe implementations in Section 6. The axiomatisation of these functions is given in the `assumes` section, of which we only show selected clauses, referring the reader to [5] for the full definition.

```
locale tables =
  fixes to-bdd-state :: 's ⇒ (bool, 'r::genr, 'v::linorder) bdd-state
  and invar :: 's ⇒ bool
  and add :: 'v ⇒ ('r ref, bool, 'v) rtree ⇒ ('r ref, bool, 'v) rtree
    ⇒ (('r ref, bool, 'v) rtree, 's) ST
  and lookup :: 'v ⇒ ('r ref, bool, 'v) rtree ⇒ ('r ref, bool, 'v) rtree
    ⇒ (('r ref, bool, 'v) rtree option, 's) SR
  and constLeaf :: bool ⇒ (('r ref, bool, 'v) rtree, 's) SR

  assumes   member-run: invar ts ⟹ (runSR (lookup v l h) ts = None)
    = (∀ r. (Node (v, r) l h) ∉ (trees (to-bdd-state ts)))
  and lookup-def: invar ts ⟹ runSR (lookup v l h) ts = Some t
  ⟹ ∃ r. runSR (lookup v l h) ts = Some (Node (v, r) l h)
        ∧ Node (v, r) l h ∈ (trees (to-bdd-state ts))
  and invar-add:invar ts ⟹ runSR (lookup v l h) ts = None
  ⟹ invar (stateST (add v l h) ts)
  and constLeaf-run:invar ts ⟹ runSR (constLeaf b) ts
    = (if b then constTrue (to-bdd-state ts) else constFalse (to-bdd-state ts))
```

Thus, there are two clauses defining the behavior of *lookup*: In case it yields *None*, the tree identified by the triple (v, l, h) is not contained in the (abstraction of the) BDD state. In the case *lookup* finds a tree *t*, it is a tree with the required attributes (v, l, h) having an undetermined reference (existentially quantified *r*).

5.3 Implementation of BDD operations

Based on the functions declared in the locale, we can now implement the functions sketched in Section 5.1.

mk tests whether its two argument trees are the same and, if this is the case, performs a simplification corresponding to the rewrite ***if*** *i* ***then*** *l* ***else*** *l* ⟶ *l*. Otherwise, it looks up the tree parameters in the table and constructs a new node in case of failure.

```
fun mk :: 'v ⇒ ('r::genr ref, bool, 'v) rtree ⇒ ('r ref, bool, 'v) rtree
  ⇒ (('r ref, bool, 'v) rtree, 's) ST where
  mk i l h =
  (if (ref-equal (l, h))
```

```
    then returnST l
  else (doST {
    (case ⟨lookup i l h⟩ of
      None ⇒ add i l h
    | Some t ⇒ returnST t) }))
```

We have factored subtree selection out into a separate function:

```
fun select :: (('a, 'v::order ∗ 'r) tree ⇒ ('a, 'v::order ∗ 'r) tree)
  ⇒ ('a, 'v::order ∗ 'r) tree ∗ ('a, 'v ∗ 'r) tree
  ⇒ ('a, 'v ∗ 'r) tree ∗ ('a, 'v ∗ 'r) tree where
 select f (t1, t2) =
  (if (levelOf t1 = levelOf t2) then (f t1, f t2)
   else (if levelOf t1 < levelOf t2 then (f t1, t2)
   else (t1, f t2)))
```

This keeps the monadically defined *app* compact:

```
function app :: (bool ⇒ bool ⇒ bool)
  ⇒ ((('r ref, bool, 'v) rtree) ∗ (('r ref, bool, 'v) rtree))
  ⇒ (('r ref, bool, 'v::linorder) rtree, 's) ST where
app bop (n1, n2) =
    (if tpair-is-leaf (n1, n2)
     then SRtoST (constLeaf (bop (leaf-contents n1) (leaf-contents n2)))
     else (doST {
         l ← app bop (select low (n1, n2));
         h ← app bop (select high (n1, n2));
         (mk (varOfLev (min-level (n1, n2))) l h) }))
```

This is the only function whose termination proof is not automatic, but still very simple: it suffices to show that *select* decreases the size of a pair of trees (defined as the sum of the sizes of the trees).

Finally, *build* is a simple recursive traversal:

```
primrec build :: 'v expr ⇒ (('r ref, bool, 'v) rtree, 's) ST
where
  build (Var i) = (doST{ mk i ⟨constLeaf False⟩ ⟨constLeaf True⟩})
| build (Const b) = SRtoST (constLeaf b)
| build (BExpr bop e1 e2) = (doST{
     n1 ← build e1;
     n2 ← build e2;
     app (interp-bbinop bop) (n1, n2) })
```

5.4 Correctness

We prove two kinds of properties, semantic and structural. They rely on a well-formedness invariant *wf_bdd_state* which expresses, among others, that the trees stored in the hash table have unique references (phrased in an object-oriented terminology: structurally equal trees are the same object), and that the hash table is closed by subtrees (if a tree is in the table, so are its subtrees).

Given this definition, we can state the semantic correctness criterion: The BDD constructed by *build* has the same interpretation as the expression it represents:

lemma *interp-evalST-build*:
wf-tables ts ⟹ *interp* (*evalST* (*build e*) *ts*) = *interp-expr e*

Furthermore, we can show that *build* establishes the structural properties required of a ROBDD: variable order and non-redundancy. For orderedness, the lemma expresses that when running *build* starting with a well-formed, ordered table, then the resulting tree is ordered (and so are the trees eventually added to the table).

lemma *build-ordered*:
⟦*runST* (*build e*) *ts* = (*t'*, *ts'*); *wf-tables ts*; *trees-prop ordered* (*to-bdd-state ts*)⟧
⟹ *trees-prop ordered* (*to-bdd-state ts'*) ∧ *ordered t'*

We can now combine this result with the canonicity of ROBDDs (lemma *canonic-robdd* in Section 3) to show that two expressions having the same interpretation give rise to two structurally equal BDDs:

lemma *canonic-build*:⟦*interp-expr e1* = *interp-expr e2*;
wf-tables ts1; *trees-prop robdd-refs* (*to-bdd-state ts1*);
wf-tables ts2; *trees-prop robdd-refs* (*to-bdd-state ts2*);
runST (*build e1*) *ts1* = (*t1*, *ts1'*);
runST (*build e2*) *ts2* = (*t2*, *ts2'*)⟧ ⟹ *struct-equal* (*t1*, *t2*)

This result is instrumental in decision procedures for propositional formulas: The BDD constructed for a valid formula is necessarily the leaf node “true”.

6 Implementation

It is now time to implement the state space along with the abstracted functions *add*, *lookup* and *constLeaf*.

We represent the state space as a couple composed of the heap containing BDDs and a hash table mapping triples (i, l, h) to BDDs:

record (*'r*, *'v*) *tables-impl* =
(*'r*, (*'r ref*, *bool*, *'v*) *rtree*) *heap* +
h-table::(*'r ref*,*bool*,*'v*) *rtree* × *'v* × (*'r ref*,*bool*,*'v*) *rtree* ⇀ (*'r ref*,*bool*,*'v*) *rtree*
constTrue :: (*'r ref*, *bool*, *'v*) *rtree*
constFalse :: (*'r ref*, *bool*, *'v*) *rtree*

To access the global variables *h_table*, *constTrue* and *constFalse*, we define monadic functions accessing the state:

definition *H-lookup* **where**
H-lookup x ≡ *SR* (λ*s*. (*h-table s x*))

definition *H-update* **where**
H-update x y ≡ *ST* (λ*s*. ((), *s*⦇*h-table* := (*h-table s*)(*x* ↦ *y*)⦈))

definition *gconstTrue* ≡ *SR* (λ*s*. *constTrue s*)

```
definition gconstFalse ≡ SR (λs. constFalse s)
```

and then we define the functions:

```
consts
  add-impl :: 'v ⇒ ('r::genr ref, bool, 'v) rtree ⇒ ('r::genr ref, bool, 'v) rtree
    ⇒ (('r ref, bool, 'v) rtree, ('r, 'v) tables-impl) ST
  member-impl :: 'v ⇒ (bool, 'v × 'r::genr ref) tree ⇒ ('r ref, bool, 'v) rtree
    ⇒ (bool, ('r, 'v) tables-impl) SR
  lookup-impl :: 'v ⇒ ('r::genr ref, bool, 'v) rtree ⇒ ('r ref, bool, 'v) rtree
    ⇒ (('r ref, bool, 'v) rtree option, ('r, 'v) tables-impl) SR
  constLeaf-impl :: bool ⇒ (('r ref, bool, 'v) rtree, ('r, 'v) tables-impl) SR

defs
   add-impl v l h ≡ doST{
    r ← alloc (λr. Node (v, r) l h);
    H-update (l, v, h) ⟨read r⟩;
    returnST ⟨read r⟩ }
  lookup-impl v l h ≡ H-lookup (l, v, h)
  constLeaf-impl b ≡ if b then gconstTrue else gconstFalse
```

It is at this point that we use the invariant component (*invar*) of the table locale for the first time: it must ensure that the heap is the inverse of the hash table and that references of nodes are their references in the heap:

```
definition invar-impl::('c, 'b) tables-impl ⇒ bool
  where invar-impl s ≡
  (∀  v l r h. (h-table s (l,v,h) = Some (Node (v, r) l h))
    ⟷ ((r, Node (v, r) l h) ∈ set (val s)))
  ∧ (∀ v l h t. h-table s (l, v, h) = Some t ⟶ (∃ r. t = Node (v, r) l h))
  ∧ (∀ r t. (r, t) ∈ set (val s) ⟶ ref t = r)
  ∧ distinct (refs s)
```

And with these definitions, we can interpret the locale *i. e.* proving the hypothesis for our implementation, and then instantiate all the functions and properties parametrized by the locale.

Our formalization [5] contains another implementation of tables, as simple lists with sequential traversal for lookup.

7 Conclusions

In this paper, we have presented first steps of a formalization and verification of a BDD package. The emphasis of this paper is more on the development method than on a high-performance algorithm. Several optimizations can be integrated without a major effort, such as memoization in function *app* and an improved representation of hash tables. We expect a garbage collector reducing the size of the hash table to provide major speed-ups. Possibly, we can adapt our verified Schorr-Waite algorithm [6] for this purpose.

We have used Isabelle's code extraction facility to produce an executable version of our algorithm in Caml and tested it on a few representative formulas, such as the valid formulas U_n defined by $x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots (x_n \Leftrightarrow (x_1 \Leftrightarrow \ldots (x_{n-1} \Leftrightarrow x_n))))$. The execution times are horrid: 2 seconds for $n = 10$, almost 180 seconds for $n = 15$. Apart from the lack of optimizations, one of the sources of inefficiency is that the Caml version explicitly manipulates the state space.

In order to definitely use pointer equality (and not just to simulate it), we have manually translated our implementation to the Scala [9] programming language (the code is available at [5]), converting monadic constructs to their imperative counterparts, erasing our reference type, replacing tests of reference equality by tests of object equality and leaving it to the Scala / Java runtime system to manage the memory. As compared to the above figures, the savings are considerable: for U_{15}, the execution time is 0.5 seconds (which, of course, is still not competitive). In the future, we hope to be able to automate this translation from Isabelle to Scala.

We also plan to make a more systematic comparison with the Isabelle Collections Framework [8], in an attempt to find commonalities between its "stateless" and our "stateful" specifications. Locales offer a good support for structuring a formal development and providing different implementations for one interface. However, in our present development, there is an unsound mixture of the specification of a theory itself (the signatures, such as *add* and *lookup*, and their properties) and elements that pertain to theory morphisms (*to_tables, invar*) that clutter up the proofs and should appear only during refinements or instantiations. We are interested in exploring alternative means of expressing interfaces and their implementations that can eventually be mapped to locales.

References

[1] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams. Technical report, IT University of Copenhagen, 1999.

[2] Clemens Ballarin. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In *Proc. 5th Intern. Conf. Mathematical Knowledge Management, MKM 2006, Wokingham, UK*, LNCS 4108, pages 31–43. Springer, 2006.

[3] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[4] Mohamed Chaabani, Mohamed Mezghiche, and Martin Strecker. Vérification d'une méthode de preuve pour la logique de description $\mathcal{ALC}$. In *Proc. 10ème Journées Approches Formelles dans l'Assistance au Développement de Logiciels*, 2010. To appear.

[5] Mathieu Giorgino and Martin Strecker. Verification of BDD algorithms by refinement of trees(formal development), 2010.

[6] Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. Verification of the Schorr-Waite algorithm - From trees to graphs. In *Proc.20th International Symposium on Logic-Based Program Synthesis and Transformation (Lopstr)*, LNCS. Springer, July 2010. To appear.

[7] Sava Krstic and John Matthews. Verifying BDD Algorithms through Monadic Interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2002), volume 2294 of LNCS*, pages 182–195. Springer, 2002.

[8] Peter Lammich and Andreas Lochbihler. The Isabelle Collections Framework. In *Proc. ITP'10*, 2010.

[9] Martin Odersky et al. An Overview of the Scala Programming Language. Technical report, EPFL, 2007.

[10] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

[11] Veronika Ortner and Norbert Schirmer. Verification of BDD Normalization. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005*, LNCS 3603, pages 261–277. Springer, 2005.

[12] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do you trust your model checker? In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD)*. Springer LNCS 1954, November 2000.

[13] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In Tobias Nipkow and Christian Urban, editors, *22nd Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLs 2009)*, LNCS 5674, Munich, Germany, 2009. Springer.

[14] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN 2000, 6th Asian Computing Science Conference*, LNCS 1961. Springer, 2000.

[15] Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, LNCS 1479, pages 461–478. Springer, September 1998.