A purely functional implementation of ROBDDs in Haskell

Jan Christiansen and Frank Huch
{jac,fhu}@informatik.uni-kiel.de

Christian Albrechts University Kiel, Germany

Abstract

This paper presents an implementation of the ROBDD data structure in Haskell. It shows that lazy evaluation can be used to improve the performance of some ROBDD algorithms. While standard implementations construct the whole structure no matter which parts are demanded we use lazy evaluation to provide a more demand driven construction. To achieve this behavior we have to relax a property that guarantees that ROBDDs contain no redundant nodes. All measurements show that relaxing causes only a small number of additional nodes. Furthermore we present an equality check implementation that performs well although it does not make use of canoicity. The canonicity is lost because of the relaxing. The equality check implementation benefits highly from laziness.

1 INTRODUCTION

A Reduced Ordered Binary Decision Diagram (ROBDD) is a data structure to represent boolean expressions. It is a compact representation that provides efficient operations for its manipulation. All BDD Package implementations, i.e., the ROBDD data structure with a couple of operations that are used in practice are written in C or C++. This paper presents the implementation of a BDD Package in Haskell.

We investigate the use of lazy evaluation to save unnecessary computations. This idea was already mentioned by Bryant who introduced ROBDDs [4]: ”One possibility would be apply the idea of ’lazy’ or ’delayed’ evaluation to OBDD-based manipulation. That is, rather than eagerly creating a full representation of every function during a sequence of operations, the program would attempt to construct only as much of the OBDDs as is required to derive the final information desired.”. Even the idea of using Haskell was brought up by Launchbury et al. [5]: ”An even more interesting question may be whether there’s some way to play off of Haskell’s strengths and take advantage of laziness.”. These two citations document the relevance behind the idea of this paper. To the best of our knowledge, despite these citations there is no approach to a lazy BDD Package implementation.

The less memory is used by an ROBDD the greater ROBDDs can be handled. If some of the ROBDD parts are not needed at all we do not have to construct them. The implementation of this idea in a strict language would be very hard. In Haskell we get this feature for free.

Even though we do not beat an up-to-date C implementation we show that the idea of lazy evaluation can be applied to the area of ROBDD manipulation. The

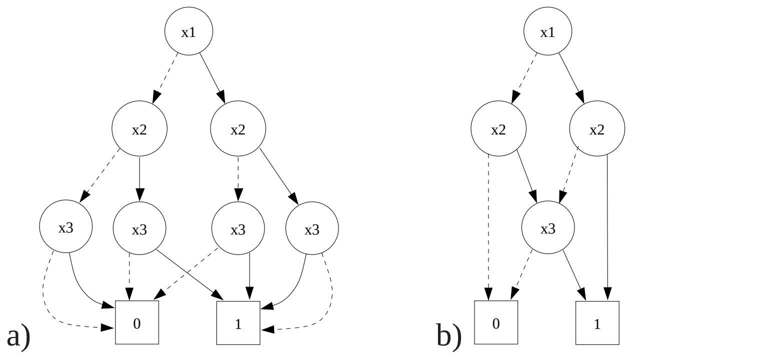


FIGURE 1. An OBDD a) and an ROBDD b) for $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

insights presented in this paper can potentially be taken back to strict languages to improve standard implementations.

2 ROBDDS

Lee introduced a data structure called Binary Decision Diagram (BDD) [11] which was popularized by Akers [1]. A BDD is a directed acyclic graph (DAG) which consists of two types of nodes. There are leaves labeled *0* and *1* representing the boolean values *false* and *true* and there are variable nodes. These nodes are labeled with boolean variables. A variable node has two successors, its low and high successor. The BDD that is rooted at the low successor represents the boolean expression that is yielded by substituting *false* for the variable. The high successor represents the boolean expression that is yielded by substituting *true*. A BDD with a fix variable order, i.e., the variables on all paths from the root node to a leaf occur in the same order is called OBDD (Ordered BDD). Figure 1 a) shows an OBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and the variable order $x_1 < x_2 < x_3$. This OBDD is the OBDD of worst case size for this expression.

In the worst case OBDDs have exponential size with respect to the number of variables. There are various OBDDs of different sizes that represent the same boolean function. Bryant introduced two properties for OBDDs and called OBDDs that satisfy these properties ROBDDs (Reduced OBDDs) [3].

An OBDD can contain two nodes with the same variable, low and high successor. For example, the two center nodes labeled x_3 in Figure 1 a) are equal in this respect. If we redirect all edges that point to one of these nodes to the other one the resulting OBDD still represents the same function. Figure 2 a) illustrates this transformation. If no node of an OBDD can be simplified in this way the OBDD satisfies the *sharing* property.

An OBDD can contain nodes whose low and high edge point to the same node. In Figure 1 a) both edges of the outermost nodes labeled x_3 point to the same node, namely the *zero* and *one* leaf respectively. The value of the whole boolean

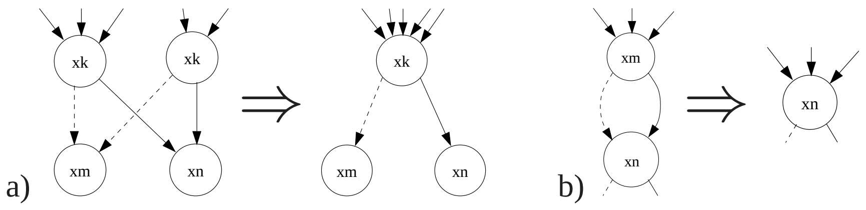


FIGURE 2. Sharing Property a) and No-Redundancy Property b)

expression is independent of the value of this variable. If we redirect all edges that point to a node like this to its successor the resulting OBDD still represents the same function. Figure 2 b) illustrates this transformation. If no node of an OBDD can be simplified this way the OBDD satisfies the *no-redundancy* property.

OBDDs that satisfy the *sharing* and the *no-redundancy* property are called ROBDDs. For a boolean function f and a fix variable order the ROBDD is the OBDD of minimal size of all OBDDs that represent f. The operation that applies these two transformations to an OBDD and yields an ROBDD is called *reduction*. Figure 1 b) shows an ROBDD for the boolean expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. While the worst case OBDD for this expression has 9 nodes the ROBDD has only 6.

Bryant proved [3] that ROBDDs are canonical with respect to a variable order. That is, for a fix variable order every boolean function is represented by exactly one ROBDD. All boolean expressions that are not satisfiable, i.e., that are the constant function *false* are represented by the same ROBDD namely the single *0* leaf. Similarly, all tautologies are represented by the same ROBDD namely the *1* leaf. Therefore, the satisfiability and the tautology check for ROBDDs are in $O(1)$. For a canonical representation the equality check is very simple because two ROBDDs are equal iff they are isomorphic.

Bryant presented operations for the efficient manipulation of ROBDDs. These operations have worst case behaviors that are linear in the number of nodes of the ROBDD they are applied to. They base on the use of memoization to process equal subtrees only once.

3 ROBDD IMPLEMENTATION IN HASKELL

The idea behind the implementation of the ROBDD data structure in Haskell is to represent a directed acyclic graph by a tree with shared sub-trees. The algebraic data type that implements this tree is called `OBDD`. There is one constructor for the nodes that takes a variable of type `Var` and one nullary constructor for each leaf.

```
data OBDD = OBDD OBDD Var OBDD
          | Zero
          | One
```

A consumer function traverses the `OBDD`. It memoizes the results of the processing of all sub-`OBDD`s. Before a sub-`OBDD` is processed the function checks whether an equal sub-`OBDD` has been processed before.

Haskell provides no mechanism to check pointer equality of two terms, i.e., to check whether two terms are shared. Because a comparison of whole sub-`OBDD`s would be inefficient we need explicit sharing in addition to the implicit sharing. The explicit sharing provides an efficient method to memoize processed sub-`OBDD`s and to preserve the *sharing* and the *no-redundancy* property.

To implement explicit sharing, we associate every node of an ROBDD with a unique identifier. These ids are integer values and we call them `NodeId`s. The `NodeId` of a node uniquely determines the structure of a sub-ROBDD within an ROBDD. That is, the root nodes of two sub-`OBDD`s have equal `NodeId`s iff these sub-`OBDD`s are structurally equal.

To check whether a node is redundant the `NodeId`s of the two successors of a node are compared. If they are equal the node is redundant. The consumer functions use these `NodeId`s for memoization. They memoize all partial results using a map called *memo map*.

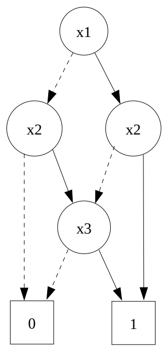
To preserve the *sharing* property we use a map that contains all constructed nodes. This mapping maps triples consisting of the `NodeId`s of low and high successor and the variable number to the `NodeId` of the node. Because of the sharing property an ROBDD contains no two nodes with the same triple but different `NodeId`s. When a node is constructed it is looked up in this map. This way two nodes with the same low and high successor and variable number get the same `NodeId`. Because the construction works bottom-up this preserves the *sharing* property. We do not only store `NodeId`s in this map but whole sub-`OBDD`s. That way equal sub-`OBDD`s are implicitly shared. Since this map is the reverse mapping of the structure of the ROBDD we refer to it as the *reverse map*. The ROBDD data type combines the `OBDD` and the *reverse map* that is implemented by the type `RevMap`.

```
data ROBDD = ROBDD OBDD RevMap
```

Each node is enriched with a `NodeId`. The leaves have the static `NodeIds` 0 and 1. We use an additional constructor called `Ref` to mark references, i.e., sub-`OBDD`s that are shared. To save memory we merge every `Ref` constructor with the outermost `OBDD` constructor of its argument and call the constructor `RefOBDD`. We refer to a `RefOBDD` constructor as a reference node and to an `OBDD` constructor as an original node.

```
data OBDD = OBDD OBDD Var OBDD NodeId
          | RefOBDD OBDD Var OBDD NodeId
          | Zero
          | One
```

References that point to leaves are not represented by `Ref` constructors. All leaves are represented by the constructors `Zero` and `One` no matter whether they are a reference or not. Haskell shares constants, i.e., all `Zero` leaves require the



```
(OBDD (OBDD Zero
            2
            (OBDD Zero 3 One 4)
            3)
      1
      (OBDD (RefOBDD Zero 3 One 4)
            2
            One
            5)
      6)
```

FIGURE 3. ROBDD and `OBDD` for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

memory of one unary constructor. The same holds for all `One` leaves.

We assure that an `OBDD` contains exactly one original node for every `NodeId`. That is, there are no two `OBDD` constructors with the same `NodeId` in an `OBDD` data structure. The original node is always the leftmost node of all nodes with the same `NodeId` in an `OBDD`. This decision is arbitrary but it has to be considered in the implementations of the consumer functions. Figure 3 shows an ROBDD and the `OBDD` data structure for this ROBDD.

We generalize the implementation of consumer functions. We present this generalization here to show how the `OBDD` data structure is used. The `fold` function for `OBDD`s is based on the standard fold function for binary trees. It uses a *memo map* to memoize partial results. The application of `fold` to a leaf yields one of the two neutral elements. We do not memoize applications to leaves because the input is constant. Therefore the computation is not expensive. The memoizing of the result would be more expensive than the computation. The `fold` function traverses the `OBDD` from left to right. If it reaches a reference node the result for this sub-`OBDD` was computed before and it is looked up in the *memo map*. If it reaches an original node it continues traversing the `OBDD` and memoizes the result for this node.

The `fold` function only looks up the `NodeId`s of reference nodes in the *memo map*. Without the `RefOBDD` constructors we would have to look up the `NodeId`s of all nodes in the *memo map*. Iff the look-up fails the node is an original node. A look-up takes a logarithmic amount of time while checking whether a node is a reference takes constant time.

It is advantageous for laziness to look up as few `NodeId`s as possible. Every look-up causes the evaluation of the `NodeId`s of all suspended inserts in the *memo map*. Furthermore the information whether a node is a reference or not saves look-ups in the *reverse map*. The prevention of these look-ups in the *reverse map* is essential for laziness. This is addressed in detail in the next section. The additional `Ref` constructors require additional memory. Even the `RefOBDD` constructors require additional memory. The outermost constructor of an original node and a reference to it cannot be shared. That is, an original node and a reference to it require the memory for the `OBDD` data structure plus the memory for one `RefOBDD` constructor. Without the `Ref` constructors we would only need the memory for the `OBDD` data structure.

```
fold :: (a → Var → a → a) → a → a → OBDD → a
fold f ez eo obdd =
  fst (fold′ emptyMemoMap obdd)
 where
  fold′ memomap Zero = (ez, memomap

  fold′ memomap One  = (eo, memomap)

  fold′ memomap (RefOBDD _ _ _ nodeId) =
    let Just v = lookupMemoMap nodeId memomap
    in
    (v, memomap)

  fold′ memomap (OBDD low var high nodeId) =
    let (lowV, lowMemomap) = fold′ memomap low
        (highV, highMemomap) = fold′ lowMemomap high
        v = f lowV var highV
    in
    (v, insertMemoMap nodeId v highMemomap)
```

The function `rOBDD` is a smart constructor for the ROBDD data structure. It adds a new node to an ROBDD. It takes a variable number, low and high successor, and the *reverse map* and yields the resulting ROBDD.

```
rOBDD :: OBDD → Var → OBDD → RevMap → ROBDD
rOBDD low var high revmap
  | getId low==getId high = ROBDD low revmap
  | otherwise =
    case lookupRevMap low var high revmap of
         Just obdd → ROBDD obdd revmap
         Nothing   → rOBDD2 low var high revmap
```

First it checks whether the node is redundant. This is the case if the `NodeIds` of the successors are equal. In this case the unchanged *reverse map* and the low successor are yielded. We could as well yield the high successor. If the `NodeIds` are not equal we look up whether a node with these successors and variable number already exists. If the look-up succeeds an `ROBDD` is yielded that contains the shared `OBDD` and the *reverse map*. All the `OBDDs` in the *reverse map* are reference nodes. Therefore the outermost constructors of all reference nodes are implicitly shared. If the look-up fails the function `rOBDD2` is applied to the arguments.

```
rOBDD2 :: OBDD → Var → OBDD → RevMap → ROBDD
rOBDD2 low var high revmap =
  let obdd = OBDD low var high (nextId revmap)
  in
  ROBDD obdd (insertRevMap low var high (toRef obdd) revmap)
```

```
apply :: (Bool → Bool → Bool) → ROBDD → ROBDD → ROBDD
negate :: ROBDD → ROBDD
restrict :: ROBDD → Var → Bool → ROBDD
anySat :: ROBDD → Maybe Binding
allSat :: ROBDD → [Binding]
evaluate :: Binding → ROBDD → Bool
(==) :: ROBDD → ROBDD → Bool
```

TABLE 1. Interface of a simple BDD Package

The function `rOBDD2` constructs an `OBDD` with a new root node and inserts this `OBDD` in the *reverse map*. The function `nextId` yields the next free `NodeId` and increases the corresponding counter in the *reverse map*.

All `OBDD`s in the *reverse map* are reference nodes. Therefore we apply the function `toRef` to the `OBDD` which replaces the outermost `OBDD` constructor by an `RefOBDD` constructor.

Table 1 shows the functions of a simple BDD Package. We have implemented all these operations for the ROBDD data structure that is presented here. The ROBDD for a boolean expression is constructed by the use of the operations `apply` and `negate`. The operation `apply` combines two ROBDDs with a boolean operator and `negate` negates an ROBDD. The ROBDD for a boolean expression can be constructed by replacing all boolean operators in the expression by appropriate applications of `apply`. The negations in the expression are replaced by applications of `negate`. The ROBDDs for the constants *true* and *false* and single variables are simple to construct. The operation `restrict` is equivalent to a substitution of a variable by *true* or *false* in the boolean expression. The consumer function `allSat` yields all satisfying bindings for an ROBDD while `anySat` yields only one. The operation `evaluate` takes a variable binding and an ROBDD and yields the boolean value that results from substituting all variables by *true* or *false* according to the given binding. Additionally, there is an equality check for ROBDDs.

4 LAZINESS

To check the laziness of this ROBDD implementation we observe which parts of the `OBDD` are evaluated when applying the function `anySat`. This function is a good check because it visits only a small number of nodes of the ROBDD. The operation `anySat` yields one satisfying binding for an ROBDD. It uses a depth first traversal to find a *one* leaf.

This implementation processes the `OBDD` from left to right. There is no rule for an `RefOBDD` constructor because it never visits a reference node. The function `anySat` visits all nodes on the path to the leftmost *one* leaf and left of it. All these nodes are original nodes. The reduction constructs the ROBDD from left to right and bottom up. The predecessor of the leftmost *one* leaf cannot be shared because the *reverse map* does not contain a node whose successor is a *one* leaf. All predecessors of this node are not shared because one of their successors is not shared. Therefore all nodes on the path to the leftmost *one* leaf are original nodes.

```
anySat :: ROBDD → Maybe Binding
anySat (ROBDD obdd _) = anySat′ obdd
 where
  anySat′ Zero = Nothing

  anySat′ One  = Just []

  anySat′ (OBDD low var high _) =
    case (anySat′ low, anySat′ high) of
         (Just path, _) → Just ((var,False):path)
         (_, Just path) → Just ((var,True):path)
```

We apply `anySat` to the ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ like it is shown in Figure 1. Figure 4 shows two observations made by the Hood observer [9, 8]. This tool provides the information which parts of a data structure are evaluated in a run of a program. Hood provides the function `observe :: String -> a -> a`. When it is applied to a `String` it behaves like the identity function and additionally records to which result its argument is evaluated. The `String` argument defines a name that is associated with this observation. At the end of the program run the observations of all `observe` applications are reported. Unevaluated parts of a data structure are represented by an underscore.

The left one results from observing the `OBDD` data structure when applying `anySat` to the corresponding ROBDD. The right observation shows the same application for an `OBDD` that does neither fulfill the *sharing* nor the *no-redundancy* property. Without the two properties only the path to the leftmost `One` leaf and all

```
(OBDD (OBDD Zero                  (OBDD (OBDD Zero
            2                                   2
            (OBDD Zero 3 One 4)                 (OBDD Zero 3 One _)
            3)                                  _)
      1                                   1
      (OBDD (RefOBDD Zero 3 One 4)        _
            _
            One
            5)
      _)                                  _)
```

FIGURE 4. Observations when applying `anySat`

parts left of it are evaluated. With the properties almost the whole `OBDD` structure is evaluated. Although `anySat` does not pattern match against the `NodeId`s all ids except for the one of the root node are evaluated.

To check whether a node is redundant we compare the `NodeId`s of the two successors of a node when it is constructed. To determine the `NodeId`s of the

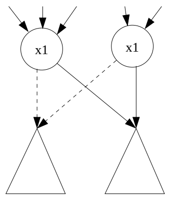


FIGURE 5. Situation before a node is shared

successors we have to check whether the successor nodes are redundant. That is, we have to compare the `NodeId`s of their successors. This results in the complete evaluation of the `OBDD` data structure if it is evaluated to head normal form. This is not surprising because the `NodeId`s determine the structure of the `OBDD` and we compare the `NodeId`s of the successors of the root node. That is, in fact we compare the structure of the `OBDD`s that are rooted at the successors of the root node.

Every check for equality of two `OBDD`s will cause the evaluation of at least the outermost constructors of the two `OBDD`s. That is, if we make any kind of redundancy check for every node the whole `OBDD` structure is evaluated if we evaluate it to head normal form. To gain any laziness in the construction of an ROBDD at all we relax the *no-redundancy* property. That is, we check whether a node is redundant for some nodes of an ROBDD but not for all. Although an ROBDD with relaxed *no-redundancy* property is not reduced anymore we carry on using the name ROBDD. We distinguish between an ROBDD with full and relaxed *no-redundancy* property. Sometimes we call an ROBDD with relaxed *no-redundancy* property short relaxed ROBDD and an ROBDD with full *no-redundancy* full ROBDD. We refer to the implementation with relaxed *no-redundancy* as relaxed implementation and to the implementation with full *no-redundancy* as full implementation.

Even without *no-redundancy* property the construction is completely strict. We have to check whether a node already exists by a look-up in the *reverse map*. This causes the evaluation of the `NodeId`s of both successors of a node. Therefore if all nodes are looked up in the *reverse map* the whole structure is evaluated just like it is the case with *no-redundancy* property.

Figure 5 illustrates the situation in which a node can be shared. Low and high edge of the right node point to the same sub-ROBDDs as low and high edge of the left node. In the `OBDD` data structure the successors of the right node are reference nodes. If one of the two successors of the right node would be no reference the node could not be shared. We would not have to look it up in the *reverse map*.

Therefore we only look up nodes whose successors are both references. The look-up of a node causes the evaluation of the `NodeId`s of the two successors. We check whether a node is redundant for nodes that are looked up in the *reverse map*. The `NodeId`s of the successors of these nodes are evaluated by the look-up anyway.

```
rOBDD :: OBDD → Var → OBDD → RevMap → ROBDD
rOBDD Zero _ Zero revmap = ROBDD Zero revmap

rOBDD One _ One revmap = ROBDD One revmap

rOBDD low var high revmap
  | isRef low && isRef high =
    if getId low==getId high
       then ROBDD low revmap
       else
        case lookupRevMap low var high revmap of
          Just obdd → ROBDD obdd revmap
          Nothing   → rOBDD2 low var high revmap
  | otherwise = rOBDD2 low var high revmap
```

We keep the *no-redundancy* property for leaves. That is, there are no redundant nodes whose successor is a leaf. Therefore all tautologies are still represented by the single *one* leaf and all unsatisfiable expressions by the single *zero* leaf. That way the complexity of the satisfiability and the tautology are still in $O(1)$.

We construct an ROBDD with relaxed *no-redundancy* property for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and check whether there is a satisfying binding for this ROBDD. We observe the `OBDD` data structure when applying `anySat` to the ROBDD. The left part of Figure 6 shows the observations of the relaxed implementation. The right part shows the observations for the full ROBDD implementation.

```
(OBDD (OBDD Zero                   (OBDD (OBDD Zero
            2                                  2
            (OBDD Zero                         (OBDD Zero
                  3                                  3
                  One                                One
                  _)                                 4)
            _)                                 3)
      1                                  1
      _                                  (OBDD (RefOBDD Zero
                                                         3
                                                         One
                                                         4)

                                               _
                                               One
                                               5)
      _)                                 _)
```

FIGURE 6. Observations for the relaxed implementation

The whole high successor of the root node is not evaluated by the relaxed implementation while it is by the full implementation. In the example with relaxed *no-redundancy* all evaluated nodes are not looked up in the *reverse map*. All these nodes are known to be no reference nodes because their low successors are original nodes.

A relaxed ROBDD has more nodes than a full ROBDD. This worsens the runtimes of some operations on this ROBDD. Besides this a relaxed ROBDD is not canonical anymore. That is, there is more than one relaxed ROBDD that represents the same boolean function. By adding a redundant node to a relaxed ROBDD we change the structure of the ROBDD but do not change the boolean function that it represents. For a canonical representation the equality check can be implemented by a check for isomorphy. The equality check for a relaxed ROBDD is more difficult.

5 EXPERIMENTAL RESULTS

Because our implementation is purely functional we get an additional logarithmic term in all operations of the BDD Package. This is caused by look-ups and inserts in the *memo* and the *reverse map*.

Figure 7 shows some measurements of the construction of an ROBDD for a boolean expression and the application of the functions `anySat` and `eval`. The function `eval` is a structural equality check for `OBDD`s. This function is used to cause the evaluation of the whole `OBDD` data structure. It is linear in the number of nodes of the ROBDD and uses no additional memory. The measurements that are provided by applications of `eval` are used to check the performance of the relaxed implementation when it cannot benefit from laziness.

We measure the time that is consumed by an application the allocated heap memory and the number of constructors that are evaluated in the `OBDD` data structure. For our performance test we use common boolean functions: e.g., `Integer 16` is the expression $(x_1 \wedge x_{17}) \vee \ldots \vee (x_{16} \wedge x_{32})$. This expression has a exponentially large ROBDD representation in the case of the canonical variable order. The boolean expression `Integer2` is the same boolean expression with another variable order. In this case the number of nodes of the ROBDD is linear in the number of variables.

The expression `Queens 8` models the eight queens problem. We use a simple coding that uses one boolean variable to indicate whether a square of the chess board is occupied by a queen or not. The expressions whose names end with the string ".cnf" belong to a library of expressions that is used for measuring SAT solvers called SATLIB [10].

For satisfiable boolean expressions, the quotient of evaluated constructors of the implementation with full *no-redundancy* and the implementation with relaxed *no-redundancy* ranges between 1531.26 for `Integer 16` and 1.17 for `Queens 8`. The number of evaluated constructors of the implementation with relaxed *no-redundancy* is less than the number of the implementation with full *no-redundancy* for all satisfiable expressions we have measured. The same holds for the time and memory consumption of the construction together with an application of `anySat`.

The number of evaluated constructors highly depends on the structure of the

Expression	Operation	No-Red.	Time	Memory	Eval. Constr.
Integer 16	anySat	relaxed	0.00	202,064	214
		full	4.36	508,343,316	327689
	eval	relaxed	4.34	512,543,932	327689
		full	4.50	515,689,380	327689
Integer2 1000	anySat	relaxed	1.70	294,052,836	505498
		full	12.78	1,838,362,908	1504498
	eval	relaxed	18.00	1,825,750,612	1504498
		full	12.98	1,837,786,380	1504498
Queens 8	anySat	relaxed	25.28	2,918,337,044	1874446
		full	32.00	3,656,326,616	2214256
	eval	relaxed	32.06	3,630,827,808	2200765
		full	32.06	3,656,438,228	2214256
uf20-02.cnf	anySat	relaxed	0.04	5,287,788	4689
		full	0.54	70,327,860	49518
	eval	relaxed	0.54	71,705,676	50930
		full	0.52	70,337,040	49518
hole8.cnf	anySat	Relaxed	20.32	2,628,758,076	1632847
		Full	20.14	2,656,013,124	1635756
	eval	Relaxed	20.10	2,628,775,708	1632847
		Full	20.44	2,656,030,756	1635756

FIGURE 7. Measurements for the construction of ROBDDs

ROBDD. For an unsatisfiable expression the number of evaluated constructors is naturally the same for the implementation with full as with relaxed *no-redundancy* property except for the additional redundant nodes. The expression `hole8.cnf` is unsatisfiable.

All measurements show that the number of redundant nodes of relaxed ROBDDs is small. In the measurements in Figure 7 only the example `uf20-02.cnf` causes the evaluation of more constructors in the relaxed implementation than in the one with full *no-redundancy*. In the examples `Queens 8` and `hole8.cnf` the number of evaluated constructors is even smaller in the relaxed than in the full implementation. This can be explained by a feature called *don't cares*. If the boolean operator $\wedge$ is applied to the *zero* leaf and an arbitrary ROBDD the result is a *zero* leaf independent of the second argument. Therefore we do not have to evaluate the second argument. The full implementation even causes the evaluation of these ROBDDs because of the redundancy checks.

6 EQUALITY CHECK

The equality check of relaxed ROBDDs can be implemented by a reduction with the full *no-redundancy* property and the isomorphy check on the results. The runtime of this implementation will be worse than the runtime of the equality check

Fst Argument	Snd Argument	Check	Time	Memory
Queens 7	uf20/uf20-02.cnf	Eq1	8.76	1,072,678,276
		Eq2	6.52	805,236,876
		Eq3	8.36	1,074,401,436
Integer 16	Integer2 1000	Eq1	23.52	2,960,983,720
		Eq2	1.38	348,603,336
		Eq3	17.86	2,528,539,232
uf20-02.cnf	uf20-02.cnf	Eq1	1.14	155,922,460
		Eq2	1.12	155,722,408
		Eq3	1.02	152,363,716
Queens 7	Queens 7	Eq1	16.24	1,989,433,408
		Eq2	16.22	1,986,515,412
		Eq3	15.56	1,996,502,316

FIGURE 8. Measurements of the Equality Check

for full ROBDDs. This implementation is completely strict. That is, even if the compared ROBDDs are not equal both ROBDDs are completely evaluated by the equality check.

We implement the equality check of relaxed ROBDDs by an application of the boolean operator $\Leftrightarrow$ and a check whether the result is the *one* leaf. This implementation has a quadratic worst case complexity in a strict language. If the compared ROBDDs are equal the complexity of this equality check is linear in the size of the ROBDD. In all other cases the operation benefits from laziness. To check whether the result of the application of $\Leftrightarrow$ is the *one* leaf it is evaluated to head normal form. This causes the evaluation of only a part of the ROBDD.

Figure 8 shows some measurements of equality checks. The first two columns state the arguments of the equality check. The third column states which equality check is used. $Eq1$ and $Eq2$ are equality checks of relaxed ROBDDs while $Eq3$ are checks of full ROBDDs. $Eq1$ uses a reduction with full *no-redundancy* and an isomorphy check while $Eq2$ uses `apply`.

The equality check that uses the isomorphy check of relaxed ROBDDs is always worse than the one of full ROBDDs. This is caused by the additional reduction. The measurements for the lazy implementation that uses `apply` are almost as good as the one of the isomorphy check which is linear in the size of the ROBDD. If two unequal ROBDDs are checked this equality check performs even better than the equality check of full ROBDDs. This is caused by the laziness of this implementation.

7 RELATED WORK

There is only one ROBDD implementation in Haskell available [2]. This was done by Jeremy Bradley in 1997. Like stated on their page this implementation is a alpha version and not very efficient. We compare the purely functional implementation using the relaxed *no-redundancy* property with this implementation. Figure 9 shows the results. The Bradley implementation cannot compete with the imple-

Expression	Operation	Implementation	Time	Memory
Queens 4	anySat	Relaxed	0.04	6,397,652
		Bradley	4.04	816,380,772
	eval	Relaxed	0.06	7,957,572
		Bradley	4.02	816,382,228
Integer 11	anySat	Relaxed	0.00	138,392
		Bradley	33.72	4,784,688,228
	eval	Relaxed	0.08	13,717,360
		Bradley	34.04	4,789,579,276
uf20-02.cnf	anySat	Relaxed	0.04	5,100,168
		Bradley	93.12	17,784,793,116
	eval	Relaxed	0.52	69,648,632
		Bradley	93.10	17,784,817,912

FIGURE 9. Comparison with the Bradley implementation

mentation presented here. This implementation is far better no matter if we use relaxed or full *no-redundancy*. The differences in the runtimes are not surprising since we use maps that support logarithmic look-up and insert operations while the Bradley implementation uses a list that supports look-up and insert operations that are linear in the number of elements. The memory usage of the Bradley implementation is surprisingly high. This implementation uses only one list where we use an algebraic data type and a map.

There are two implementations of interfaces to BDD packages using the foreign function interface of the GHC. The first was presented in 1999 by Day, Launchbury and Lewis [5]. We use a user interface for the construction of boolean expressions which is very similar to the one presented in that work. They use this interface to bind the CMU Long BDD Package to Haskell. Their interface is referentially transparent which allows the user to ignore the details of the imperative implementation.

The other binding of a BDD Package is HBDD [6]. This is a Haskell interface that can be used with the CMU Long BDD Package, too. Bindings to CUDD and BuDDy are planned. HBDD is used in MCK [7] a model checker for the logic of knowledge written in Haskell. We compare our implementation with the HBDD binding. This binding is more up-to-date. We use optimizations in this measurements. The C implementation is highly optimized and therefore we want to bring out the best in our implementation. We do not consider the internal implementation of the BDD Package. Therefore we are unable to explain the differences in the measurements considering the internals anyway.

It is not completely clear to this moment whether the memory that is allocated by the C program is reflected in the values of the profiling. The HBDD implementation consumes less time and memory than the implementation presented here in

Expression	Operation	Implementation	Time	Memory
Queens 8	anySat	Relaxed	8.14	1,413,390,596
		HBDD	1,04	1,476,988
	eval	Relaxed	10.36	1,754,746,240
		HBDD	1,04	1,377,600
Integer 19	anySat	Relaxed	0.00	166,972
		HBDD	0,00	69,916
	eval	Relaxed	10.54	2,223,653,380
		HBDD	0,00	58,256
Integer2 1800	anySat	Relaxed	2.92	587,783,116
		HBDD	0,04	6,103,648
	eval	Relaxed	21.82	3,138,158,752
		HBDD	0,04	4,080,732

FIGURE 10. Comparison with the HBDD implementation

all measurements. On the one hand the C implementation uses lots of refinements. A major one is the variable reordering. The best example for the use of variable reordering is the `Integer` expression. This expression is of exponential size with the canonical variable order that is used by our implementation. The best order causes `Integer` to be linear in the number of variables like `Integer2` which uses this optimal variable order. On the other hand C is more efficient than Haskell.

8 CONCLUSION

This paper demonstrates that even a complex data structure like ROBDDs can benefit from laziness. We improve the performance of the operations on ROBDDs with respect to time and memory consumption. The *no-redundancy* property of ROBDDs causes the evaluation of the whole ROBDD when we apply an operation to it.

Relaxing the *no-redundancy* property is an adequate answer to this problem. Experiments show that the disadvantages of relaxing are small or not existent. The number of redundant nodes is very small for all examples we have measured. More or less all operations benefit from laziness. Relaxing the *no-redundancy* property is an elementary modification of the ROBDD data structure. This is in fact a variation of the data structure and not an implementation detail.

One disadvantage of the implementation presented here is that it does not use one *reverse map* for all ROBDDs. This is an extension that was proposed shortly after the publication of the ROBDD data structure. If we use one *reverse map* the number of nodes is reduced because ROBDDs share equal sub-ROBDDs. Additionally the performance of the construction is improved. The `apply` operation memoizes the application of a boolean operator to a pair of sub-ROBDDs. With one *reverse map* all applications of `apply` with a specific boolean operator can share one memoization map.

The use of one *reverse map* would require use to pass the *reverse map* from one application of `apply` to another. Therefore it would require a monadic like implementation of the construction of an ROBDD. First tests showed that this extension does not cooperate with laziness. That is, we lose laziness in the construction if we use it. One look-up in the *reverse map* would cause the evaluation of all `NodeId`s of the nodes that were constructed so far.

Apart from the advantages in the semantics it is very hard to benefit from laziness if this should exceed the standard examples like infinite data structures. It is very easy to destroy the laziness of an algorithm. Furthermore it is very difficult to locate the origin because of the complexity of lazy evaluation. Its unlikely that an algorithm is implemented without bothering about laziness and benefits from it as a side effect. There are no tools that explicitly support the design of lazy algorithms.

We hope that this paper is the starting point for further research on the benefits and disadvantages of lazy evaluation for the efficiency of algorithms. Still today eight years after the definition of the Haskell 98 Standard this issue is highly up-to-date.

REFERENCES

[1] Akers, S., *Binary Decision Diagrams*, IEEE Trans. actions on Computers **C-27** (1978), pp. 509–516.

[2] Bradley, J., *Binary decision diagrams - A functional implementation* (1997), http://www.cs.bris.ac.uk/ bradley/publish/bdd/.

[3] Bryant, R. E., *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Comput. **35** (1986), pp. 677–691.

[4] Bryant, R. E., *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Comput. Surv. **24** (1992), pp. 293–318.

[5] Day, N. A., J. Launchbury and J. Lewis, *Logical abstractions in Haskell*, in: *Proceedings of the 1999 Haskell Workshop* (1999).

[6] Gammie, P., *A Haskell binding to Long's BDD library*, http://www.cse.unsw.edu.au/ mck/.

[7] Gammie, P. and R. van der Meyden, *MCK: Model checking the logic of knowledge*, in: *Proceedings of the 16th International conference on Computer Aided Verification, CAV*, 2004, pp. 479–483.

[8] Gill, A., *Debugging haskell by observing intermediate data structures* (2000).

[9] Gill, A., *The haskell object observation debugger* (2000), http://www.haskel.org/hood/.

[10] Hoos, H. H. and T. Stützle, *SATLIB: An online resource for research on SAT*, in: *SAT 2000* (2000), pp. 283–292, http://www.satlib.org.

[11] Lee, C., *Representation of switching circuits by binary decision diagrams*, Bell System Technical Journal **38** (1959), pp. 985–999.