Christian-Albrechts-Universität zu Kiel

Diploma Thesis

A purely functional implementation of ROBDDs in Haskell

Jan Christiansen

February 9th, 2006

Institute of Computer Science and Applied Mathematics
Programming Languages and Compiler Construction

Supervised by:
Prof. Dr. Michael Hanus
Dr. Frank Huch

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angebenen Hilfsmittel verwendet habe.

Kiel,

_____

Contents

1 Introduction

A Reduced Ordered Binary Decision Diagram (ROBDD) is a data structure to represent boolean expressions. This is a compact representation that provides efficient operations to manipulate the expression. All implementations of a BDD Package, i.e., the ROBDD data structure with a couple of operations that are used in practice are written in C or C++. The goal of this work is to implement the ROBDD data structure and the most important operations in Haskell [20]. Haskell is a lazy, purely functional programming language, that provides algebraic data types, static typing, higher-order functions and polymorphism. This paper discusses the design choices that where made in the implementation.

A main aspect that is observed in this paper is the use of lazy evaluation to save unnecessary computations. This idea was already mentioned by Bryant who introduced ROBDDs [10]: "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. [11]: "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 thesis. Despite these citations there is no approach to an ROBDD implementation of this kind.

The main goal of ROBDD implementations is to save memory. The less memory is used by an ROBDD the greater ROBDD 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. For free is not quite correct because the mechanisms that provide the lazy evaluation cost memory.

The aim of this thesis is not to beat any C implementation. One goal is to beat the only present purely functional Haskell implementation. Besides this we provide an implementation that makes no use of laziness and compare this one with an implementation that focuses on using laziness to save unnecessary computations. Even though we do not beat an up-to-date C implementation we show that the idea of lazy evaluation can be applied to this area of ROBDD manipulation. The insights presented in this paper can potentially be taken back to strict languages to improve the standard implementations.

1.1 Lazy Evaluation

The run of a functional program is the evaluation of an expression. There are multiple strategies to evaluate an expression. There are two distinctions in the evaluation strategy that decide which part of an expression is evaluated first. We have to decide whether to evaluate the outermost or the innermost expression first. Second we have to decide whether the leftmost or the rightmost expression is evaluated first.

There are two special evaluation strategies, leftmost innermost (LI) and leftmost outermost (LO). Strict functional languages use a leftmost innermost reduction. All arguments have to be evaluated before the function is evaluated, i.e., before a function application is replaced by the definition of the function. Non strict functional languages use a leftmost outermost evaluation strategy. It evaluates functions before it evaluates the arguments of the function. This evaluation strategy is computational complete. That is, if there is any evaluation strategy that yields a result the leftmost outermost strategy yields it, too. Figure 1.1 shows an example of a leftmost outermost and a leftmost innermost reduction of the expression `head ([1,2] ++ [3.4])`. The function `head` yields the first element of a list. The function `(++)` is the concatenation of lists. The definitions of both can be found in the Haskell Report [20]. The leftmost outermost reduction pre-

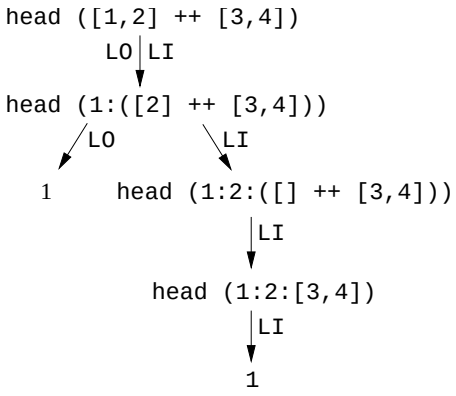

Figure 1.1: LI and LO reduction of `head ([1,2] ++ [3,4])`

vents unnecessary evaluations. In the example the outermost reduction only evaluates the head of the list while the innermost reduction causes the evaluation of the whole list. By the term laziness in this paper we denote the fact that parts of an expression are not evaluated because of lazy evaluation, i.e., that parts of a data structure are not constructed or function applications are not evaluated.

Figure 1.2 shows the reduction of `square (23 + 42)` using a leftmost outermost strategy. The function `square` yields the square of a number by multiplying the number with itself. This example shows a problem of the leftmost outermost strategy. The expression `23 + 42` is evaluated twice. This is caused by the copying of the argument of `square`.

The term lazy evaluation describes an evaluation strategy that uses a leftmost outermost strategy and prevents that an expression is evaluated twice. This evaluation is implemented by a graph. This graph takes care that an expression is only evaluated

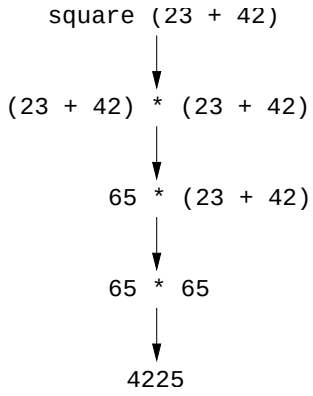

Figure 1.2: LO reduction of `square (23 + 42)`

once. Every node of this graph represents an expression. We do not copy expressions but add a pointer to the node that represents this expression. If an expression is evaluated the node that represents this expression is updated to the value that is yielded by the evaluation. If we demand the evaluation of this term a second time we do not evaluate it but just look up the value by dereferencing the pointer. A node is represented by a position in the heap memory. If two expressions are represented by pointers to the same place in the heap, i.e., pointers to the same node in the graph, the expressions are said to be shared. Shared expressions have two characteristics. First they are evaluated only once. Second the expressions do only require the memory for one expression in the heap. John Launchbury has formalized what was sketched here to a semantics also known as Launchbury semantics [21].

1.2 Boolean Expressions

Boolean expressions are often used in computer science. Mostly they are expressed either in disjunctive normal form (DNF) or in conjunctive normal form (CNF). The satisfiability check for a boolean expression in an arbitrary form is NP-complete. Determining whether a formula in CNF is satisfiable is still NP-complete, even if each clause is limited to at most three literals. This problem is known as 3-SAT. For boolean expressions in DNF satisfiability is decidable in polynomial time.

To check whether a formula is a tautology we can use the satisfiability check. A formula is a tautology iff its negation is not satisfiable. The negation of a boolean formula in CNF is a boolean formula in DNF and vice versa. Therefore the tautology check is co-NP complete for DNFs and decidable in polynomial time for CNFs. This seems to suggest to check for tautology in CNF and satisfiability in DNF but the conversion from one to another is exponential in the number of variables in the worst case.

This leads to another normal form that is called INF (*if-then-else* normal form). With two additional conditions this normal form supports tautology and satisfiability check in $O(1)$. To generate the INF of a boolean expression we use the *Shannon Expansion*.

We first introduce an operator called *if-then-else*. This expression is read "if x then y_0 else y_1".

$$x \rightarrow y_0, y_1 = (x \wedge y_1) \vee (\neg x \wedge y_0)$$

A boolean expression in INF consists only of the *if-then-else* operator and the constants *true* and *false*. The *Shannon Expansion* expresses the relation between a boolean expression t over one variable x and this expression in INF.

$$t \equiv x \rightarrow t[x \mapsto 0], t[x \mapsto 1]$$

The expression $t[x \mapsto 0]$ denotes the substitution of all occurrences of x in t by 0, i.e., *false*. By iterated use of this statement we can generate an INF for every boolean expression. We have to use the *Shannon Expansion* once for every variable in the expression.

1.3 Decision Trees

We can represent a boolean expression in INF by a Decision Tree. A boolean expression in INF is a term that is composed of three constructors. The two constants *true* and *false* and the three-ary constructor *if-then-else*. If we look upon this term as a tree we get the corresponding Decision Tree. A Decision Tree is a binary tree where each node represents the use of the *Shannon Expansion*. The node is labeled with the variable that is substituted. The left successor represents the Decision Tree for the expression where this variable is substituted by *false*. This successor is also called the low successor. The right or high successor represents the substitution by *true*. The leaves are labeled one and zero for the boolean constants *true* and *false* respectively.

If we want to construct a Decision Tree out of a boolean expression we have to choose a variable that is substituted on every application of the *Shannon Expansion*, i.e., we have to choose the x in the *Shannon Expansion*. If we use the same order of variables for every path from the root to a leaf the tree is called Ordered Decision Tree. Figure 1.3 shows an Ordered Decision Tree for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. On all paths from the root to a leaf the variables occur in the order x_1, x_2, x_3.

Implementation of Decision Trees

If we implement a Decision Tree in Haskell in fact we implement INF terms. In a functional programming language a tree data structure is implemented by an algebraic data type. Each node of the Decision Tree takes three arguments, its low successor, its high successor and its variable. Additionally we need two leaves for the constants *true* and *false* called `Zero` and `One`. We use the type synonym `Var` for the variables. The concrete implementation is not important here.

```
data DT = DT DT Var DT
        | Zero
        | One
```

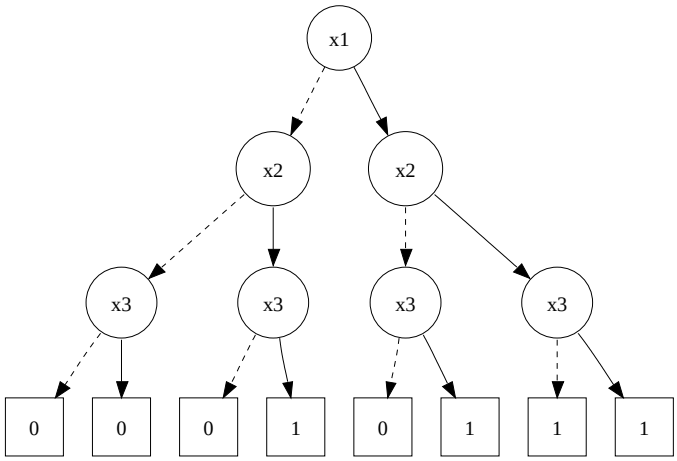

Figure 1.3: A Decision Tree for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

We define a function `build` that constructs a Decision Tree out of a boolean expression. This is done by iterative application of the *Shannon Expansion*. We define a type `BExp` that represents boolean expressions. The constants *true* and *false* are represented by the nullary constructors `BFalse` and `BTrue`. We assume a `substitute` function for these boolean expressions. This function additionally simplifies the expression as far as possible. That is, if the boolean expression contains no more variables `substitute` yields `BFalse` or `BTrue`. Without this simplification `build` would not terminate.

```
build :: BExp → DT
build bexp = build' 1 bexp


build' :: Var → BExp → DT
build' _ BFalse = Zero

build' _ BTrue  = One

build' var exp =
  let lowExp = substitute exp var BFalse
      highExp = substitute exp var BTrue
      low = build' (succ var) lowExp
      high = build' (succ var) highExp
  in
  DT low var high
```

The definition of `build` is straight forward. The termination cases are applications to `BFalse` and `BTrue` respectively. If the boolean expression is not a constant we substitute the current variable by `BFalse` and `BTrue` and apply `build'` to the results. These applications yield the low and high successor of the current node. We have to use a type that is a member of the type class `Enum`. This class provides the function `succ` which

yields the successor of its argument. This function is used to determine the next variable that is used for the *Shannon Expansion*. We use the type `Int` for the variables. This type provides an efficient `succ` implementation. Later we need an efficient comparison of two variables. This is provided by `Int` as well. We have to start the construction with the substitution of the smallest variable. We determine that all variables in a boolean expression are greater or equal one and use variable one for the first *Shannon Expansion*.

1.4 Reduced Ordered Binary Decision Diagrams

Decision Trees are not the best representation for boolean expressions because their size grows exponential in the number of variables. Lee introduced a data structure called Binary Decision Diagram (BDD) [22] which was popularized by Akers [2].

A BDD is a directed acyclic graph (DAG). This graph consists of two types of nodes. There are leaves labeled 0 and 1. The *zero* leaf represents *false* and the *one* leaf *true*. The second type of nodes are variable nodes. These nodes are labeled with a variable number. A variable node has two successors, the low and the high successor. A BDD with a fix variable order, i.e., the variables on all paths from the root to a leaf occur in the same order is called OBDD (Ordered BDD). A BDD is a compressed form of a Decision Tree because equal sub-trees may be shared. At least all *zero* and all *one* leaves are shared. Figure 1.4 shows an OBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. This OBDD is the OBDD of worst case size for this expression.

OBDDs have exponential size in respect to the number of variables in the worst case, too. They are smaller than Decision Trees but not guaranteed to be of minimal size. Bryant introduced two properties for OBDDs and called OBDDs that satisfy these properties ROBDDs (Reduced OBDD) [8]. For a boolean function f and a fix variable order the ROBDD is the OBDD of minimal size of all OBDDs that represent the function f.

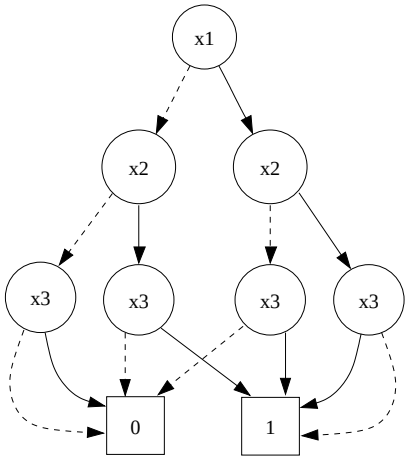

Figure 1.4: An OBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

An OBDD can contain two nodes with the same variable, low and high successor. In Figure 1.4 the two center nodes labeled x_3 have the same variable, low and high successor. All edges that point to one of these nodes are redirected to the other one. If a node cannot be simplified by this rule it satisfies the *sharing* property. Figure 1.5 shows an example of the application of this rule. If no node of an OBDD can be simplified by this rule the OBDD satisfies the *sharing* property.

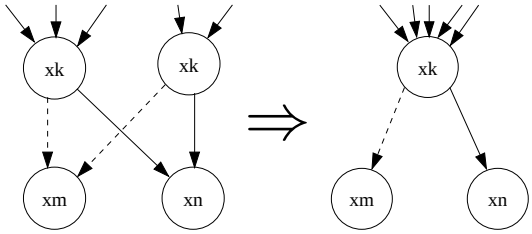

Figure 1.5: The Sharing Property

An OBDD can contain nodes whose low and high edge point to the same node. In Figure 1.4 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 expression is independent of the value of this variable. All edges that point to a node whose low and high edge point to the same node are redirected to one of the successors of this node. If a node cannot be simplified by this rule, it satisfies the *no-redundancy* property. Figure 1.6 shows an example of the application of this property. If no node of an OBDD can be simplified by this rule the OBDD satisfies the *no-redundancy* property.

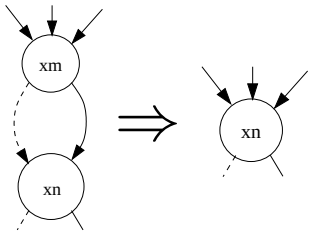

Figure 1.6: The No-Redundancy Property

ROBDDs satisfy the *no-redundancy* and the *sharing* property. The operation that applies these two rules to an OBDD and yields an ROBDD is called reduction. Figure 1.7 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 Decision Tree for this expression has 15 nodes the worst case OBDD has 9 and the ROBDD has 6.

Bryant proved [8] 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. It is important to talk about boolean functions and not about boolean expressions because there are many boolean expressions that represent the same boolean function. For

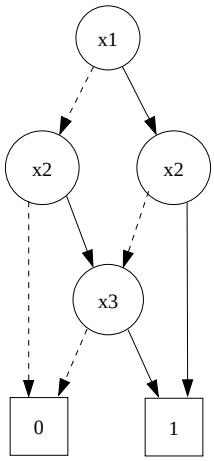

Figure 1.7: An ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

example there are many boolean expressions that represent the constant function *false* but only one function and therefore one ROBDD.

All boolean expressions that are not satisfiable, i.e., that are the constant function *false* are represented by the same ROBDD. The same holds for the constant *true*, i.e., all tautologies. Therefore the satisfiability and the tautology check are in $O(1)$ for ROBDDs. The ROBDD data structure satisfies all properties that were asked for in Section 1.2. Another advantage of the canonical representation is that the equality check becomes very simple because two equal ROBDDs are isomorphic.

Furthermore Bryant proved [8] that any function graph for a function f that is not isomorphic to the corresponding ROBDD has more nodes. Besides these proofs Bryant presented operations for the efficient manipulation of ROBDDs. These operations have worst case behaviors that are linear or quadratic in the number of nodes of the ROBDDs they are applied to. We present these operations in the next section.

The size of an ROBDD and therefore the efficiency of the operations on this ROBDD highly depends on the variable order. For example the expression $(a_1 \wedge b_1) \vee \cdots \vee (a_n \wedge b_n)$ with the variable order $a_1 < b_1 < \cdots < a_n < b_n$ is represented by an ROBDD with $2 * (n + 1)$ nodes. The same expression with the order $a_1 < \cdots < a_n < b_1 < ... < b_n$ is represented by an ROBDD with $2^{(n+1)}$ nodes. Figure 1.8 shows the ROBDDs for $n = 3$ for both orders.

It is NP-hard to find an optimal order but there are many approaches to find a good variable order. We do not discuss the choice of a variable order in this paper. The implementations that are presented in this paper all use the canonical variable order, i.e., $x_1 < \cdots < x_n$.

Today ROBDDs are widely used in computer science. They are used in VLSI CAD, in Model Checking, for representing Relations and many other domains where fast boolean expression manipulation is needed. The worst case size of an ROBDD is still exponential in the number of variables but the ROBDD representations for most expressions that are used in practice are reasonable small.

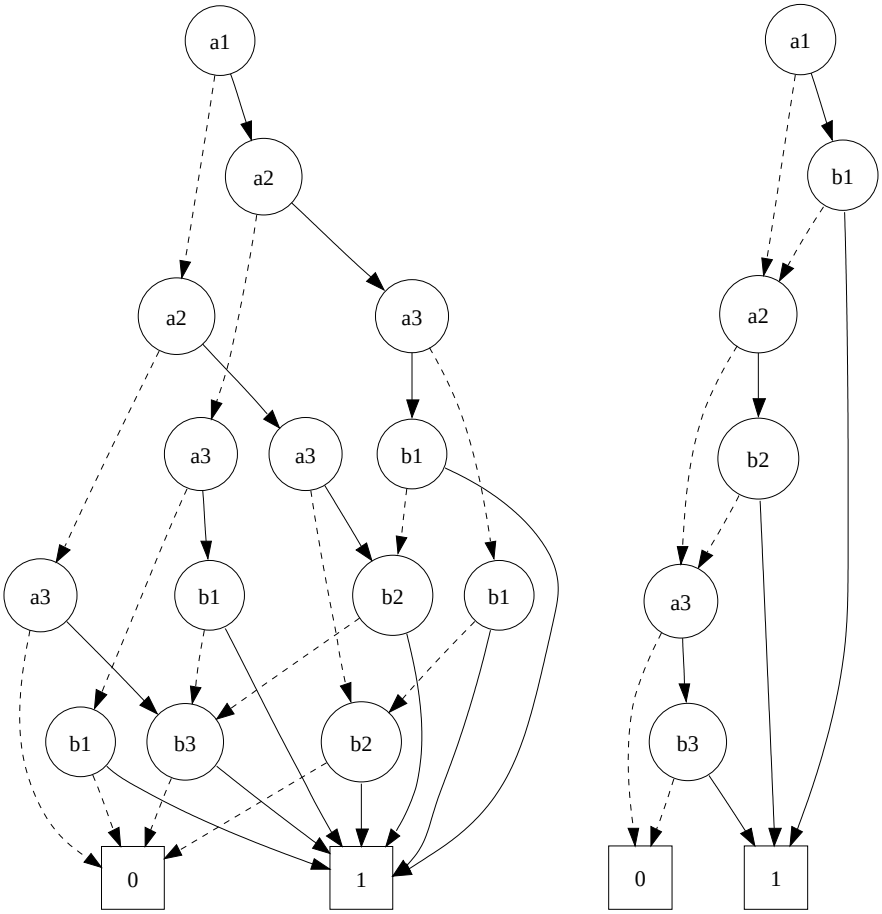

Figure 1.8: ROBDDs for the expression $(a_1 \wedge b_1) \vee \cdots \vee (a_3 \wedge b_3)$ using the order $a_1 < b_1 < \cdots < a_3 < b_3$ and $a_1 < \cdots < a_3 < b_1 < ... < b_3$

1.5 BDD Package

This section presents the most important operations on ROBDDs and their complexities in an imperative framework. All operations are reviewed when their Haskell implementation is presented. BDD Packages, i.e., the ROBDD data structure and a couple of operations are usually written in C or C++. The widest known packages are CUDD [30] and the CMU BDD library [23].

The main idea behind the efficient implementation of the operations on ROBDDs is the use of memoization. The memoization is used to assure that equal sub-ROBDDs are processed only once. The first application to an sub-ROBDD is memoized and all other applications to the same sub-ROBDD are looked up. The running times of most of the operations are linear in the number of nodes of the ROBDD. Without this memoization the running times would be exponential in the number of variables. For the memoization we need keys for the insert and the look-up of the partial results. Each node of an ROBDD is associated with an integer value. We name these integer values

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

Table 1.1: Interface of a simple BDD Package

`NodeId`s.

All running times that are presented here assume an implementation that uses destructive updates. This provides the benefit that look-ups and inserts of integer values in a map-like structure are in $O(1)$.

Table 1.1 shows the operations that are part of a simple BDD Package. Packages like CUDD support a variety of other functions for example for variable reordering. The `build` operation that constructs an ROBDD out of a boolean expression is not part of a standard BDD Package because its running time is exponential in the number of variables in the boolean expression. We use this function to present the basic idea of the ROBDD construction and to investigate the laziness of the reduction of an OBDD to an ROBDD. The alternative construction is much more complex and therefore not reasonable for an introduction.

The variables f, g, and h always denote ROBDDs. The function $|\cdot|$ takes an ROBDD and yields its size, i.e., the number of nodes in the ROBDD. The variables n and m denote the number of variables in an ROBDD. If it denotes the number of variables in the expression it is explicitly mentioned.

```
evaluate :: Binding -> ROBDD -> Bool
```

The operation `evaluate` takes a variable binding and an ROBDD and yields the boolean value that results from substituting all variables by *true* and *false* according to the given binding. This operation starts at the root and takes the low and high successor respectively at every node till it reaches a leaf. If this is a *zero* leaf the operation yields `False` otherwise `True`. In an imperative implementation we get the low and high successor of a node in $O(1)$. Thus `evaluate` has a worst case running time in $O(n)$ where n is the number of variables.

```
anySat :: ROBDD -> Maybe Binding
```

The operation `anySat` yields a variable binding that satisfies the corresponding boolean expression if one exists. This binding contains only the relevant variables. If no such binding exists `anySat` yields `Nothing`. The only ROBDD that is not satisfiable is the single *zero* leaf, i.e., the ROBDD that represents the constant boolean function *false*. The function `anySat` uses a depth first traversal to find a *one* leaf. If an ROBDD is not the *zero* leaf we know that it is satisfiable. To find a variable binding we have to

check whether one of the successors is the *zero* leaf. In this case we follow the other successor. Because of the *no-redundancy* property there is no node whose successors are both unsatisfiable. Every variable occurs at most once on a path from the root to a leaf. We have to visit at most n nodes where n is the number of variables in the boolean expression. Therefore `anySat` is in $O(n)$.

```
allSat ::  ROBDD -> [Binding]
```

The function `allSat` is similar to `anySat`. It yields all variable bindings that satisfy the expression leaving out irrelevant variables. The return type is a list of variable bindings. If none exists `allSat` yields the empty list. This function is rarely used because its result can contain exponentially many elements with respect to the number of variables. The worst case running time of `allSat` is $O(n|S_f|)$ where $|S_f|$ denotes the size of the satisfying set of the ROBDD f. The result of `allSat` has at most $n|S_f|$ elements. Just printing the result is in $O(n|S_f|)$. In the worst case we have to add all variables to every element of the satisfying set. That is, if the concatenation of two lists and adding to the front are in $O(1)$ the worst case running time of `allSat` is in $O(n|S_f|)$. This cannot be improved by memoization because it is also a lower bound.

```
satCount ::  ROBDD -> Int
```

The function `satCount` calculates the number of bindings that satisfy the ROBDD. Let n be the greatest variable number in the expression. The application of `satCount` yields the number of variable bindings consisting of the variables x_1 to x_n that evaluate the ROBDD to `True`. That is, `satCount` considers variables that are left out in the ROBDD. The result of `satCount` is not equal to the length of the result of `allSat` because `allSat` leaves out irrelevant variables.

Let the greatest variable in the expression for the ROBDD in Figure 1.9 be x_6. There is one satisfying binding for a *one* leaf and none for a *zero* leaf. There are two variable bindings that satisfy the sub-ROBDD rooted at the node labeled x_4 namely $[(x_4, \texttt{True}), (x_5, \texttt{False})]$ and $[(x_4, \texttt{True}), (x_5, \texttt{True})]$. The low successor of the node labeled x_4 yields none, the high successor yields one binding. The variables that are left out, in this example x_5, can be set arbitrarily. If we leave out k variables there are 2^k bindings that set the left out variables arbitrarily.

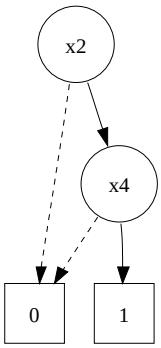

Figure 1.9: An ROBDD for the expression $x_2 \wedge x_4$

The function `getLow` yields the low, `getHigh` the high successor of a node. The function `var` yields the variable number of a node. The number of variables that are left out between a node and its low successor is $var(getLow(f)) - var(f) - 1$. That is, we get $2^{var(getLow(f))-var(f)-1} \cdot satCount(getLow(f))$ bindings consisting of the variables $var(f)$ to x_n for the low successor of a non-terminal f. The same holds for the high successor. Therefore a non-terminal f gets the value $2^{var(getLow(f))-var(f)-1} \cdot satCount(getLow(f)) + 2^{var(getHigh(f))-var(f)-1} \cdot satCount(getHigh(f))$. All leaves get the variable number of the greatest variable in the expression. This way the left out variables at the leaves are set arbitrarily.

The application of this formula to the example yields four. This does not consider the variables that are left out at the root. That is, we have to add two to the power of the number of variables that are left out at the root times the result of the application of `satCount` to the ROBDD. Altogether we get eight for the example.

This formula was published by Bryant [8]. By using memoization we get a worst case running time in $O(|f|)$ where f is the ROBDD we are applying `satCount` to.

```
restrict :: ROBDD -> Var -> Bool -> ROBDD
```

The `restrict` operation on an ROBDD is equivalent to the substitution of a variable by *false* or *true* in the boolean expression. On the ROBDD the restriction replaces all nodes with a corresponding variable by its low and high successor respectively. This operation uses memoization to restrict equal sub-trees only once and is in $O(|f|)$. If `restrict` is applied to a sub-tree that has been processed already the result is looked up.

```
apply :: (Bool -> Bool -> Bool) -> ROBDD -> ROBDD -> ROBDD
```

The `apply` operation combines two ROBDDs by a boolean operator. This function has a worst case running time in $O(|f||g|)$ where f and g are the two ROBDDs. This running time is achieved by memoizing all applications to two sub-ROBDDs. There are at most $|f||g|$ applications of `apply` to a pair of nodes.

```
negate :: ROBDD -> ROBDD
```

The function `negate` on ROBDDs is equivalent to the not function on a boolean expression. It replaces all references to *zero* leaves in the ROBDD by references to *one* leaves and all references to *one* leaves by references to *zero* leaves. By using memoization we achieve a running time in $O(|f|)$. By using complement edges like explained in section 7.2 the running time can be improved to $O(1)$.

```
(==) :: ROBDD -> ROBDD -> Bool
```

The function `(==)` is the equality check on ROBDDs. Because ROBDDs are a canonical representation of boolean functions equal ROBDDs are isomorphic. The equality check `(==)` is a check for isomorphy that is implemented by a simultaneous traversal of the two ROBDDs. Its running time is in $O(min\{|f|, |g|\})$ where f and g are the two ROBDDs. If we use an extension that is introduced in Section 7.1 the equality check is in $O(1)$.

2 Test Toolbox

This chapter introduces the test toolbox that is used in this paper. We introduce example expressions and tools that are used for the measurement of the laziness and the efficiency of the presented implementations.

2.1 Boolean Expressions

This section introduces some boolean expressions that are used in this paper. We call expressions that are exponential in the number of variables *hard* expressions. These expressions are the worst case of the ROBDD operations. Section 1.4 introduced the integer expression whose `ROBDD` representation has $2 * (n + 1)$ or $2^{(n+1)}$ nodes depending on the variable order. We call the version of this expression that has exponential many nodes `Integer` and the one that has linear many nodes `Integer2`. If we use variable reordering the ROBDD for this expression has linear many nodes in the number of variables, too.

There are expressions that have exponential many nodes in the number of variables for all variable orders. One example is the hidden weighted bit function *HWB* that was introduced by Bryant [9]. For each variable order the size of the ROBDD that represents this function is in $O(n2^{0.5n})$ [5]. It is defined as

$$HWB(x_1, \ldots, x_n) = \begin{cases} x_s \text{ if } s \geq 1, \\ 0 \text{ if } s = 0. \end{cases} \quad \text{where } s = sum(x_1, \ldots, x_n)$$

Besides the *hard* expressions we use *SAT* expressions, i.e., expressions that define a satisfiability problem. One example is the eight queens problem. We use a simple representation for this problem. We model every field of the chess board by one boolean variable. Iff this variable is *true* the corresponding field is occupied by a queen. We call this expression `Queens n` where the `n` specifies the number of queens.

There is a library of expressions that are used for measuring SAT solvers. This library is called SATLIB [19]. It uses a CNF format [31] for the definition of the expressions. This library provides a couple of expressions. The names of all the expressions that belong to this library end with the string ".cnf". More information on theses expressions can be found on the SATLIB homepage.

2.2 Checking for Laziness

Most tools that are concerned with the evaluation of Haskell programs abstract from the lazy evaluation. For example the hat debugger pretends a strict evaluation order for

producing a trace that is easier to understand by the user. We observed this to be a problem. To find the origin of an unexpected evaluation the strict evaluation does not help.

To check the laziness of our algorithms we use the Hood observer [16, 15]. 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 are represented by an underscore.

Here is a simple example and the observations that result from applying the function `main`. The function `print` causes the evaluation of all elements of the list.

```
list :: [Int]
list = [1,2,3,4,5]

main = print (observe "list" list)

Main> main
[1,2,3,4,5]

>>> Observations <<<

list
  (1 : 2 : 3 : 4 : 5 : [])
```

In the example below we add the application of the function `length`. The elements of the list are not evaluated. The function `length` only demands the evaluation of the list data structure. The unevaluated elements are represented by underscores.

```
list :: [Int]
list = [1,2,3,4,5]

main = print (length (observe "length" list))

Main> main
5

>>> Observations <<<

length
  (_ : _ : _ : _ : _ : [])
```

We can use Hood to observe the applications of a function. The observations of a function are shown in the form of a mapping from the arguments to the results. We observe the partial application `(+ 1)` to a list of numbers. The function `inc` is applied five times in this example once to every element in the list.

```
inc = observe "inc" (+ 1)

main = print (map inc [1,2,3,4,5])

Main> main
[2,3,4,5,6]

>>> Observations <<<

inc
  { \ 5  → 6
  , \ 4  → 5
  , \ 3  → 4
  , \ 2  → 3
  , \ 1  → 2
  }
```

If we apply a `take 3` to the result of `map` two of the applications of `inc` are never evaluated and therefore are not shown in the observations.

```
main = print (take 3 (map inc [1,2,3,4,5]))

Main> main
[2,3,4]

>>> Observations <<<

inc
  { \ 3  → 4
  , \ 2  → 3
  , \ 1  → 2
  }
```

2.3 Measurements

The measurements that are presented throughout this paper always state at least two values namely the time and the total heap memory that was consumed by the operation. These two values are measured using the profiling that comes with the Glasgow Haskell Compiler [29]. The memory usage is more significant than the running time. The running time depends on the scheduling of the processes. We present the running times to give the reader an impression of the efficiency of the implementations. We do not profile the heap memory usage. That is, we do not check how much heap memory is used at one time. We only check the total amount of heap memory that is used. This problem is addressed in Chapter 7.

The final implementation of the ROBDD data structure uses an algebraic data type. Most of the measurements state the number of constructors of this data type that are

evaluated. This number is an indication for the laziness of an implementation. A strict implementation evaluates all constructors no matter which parts of the ROBDD are needed to compute the final result. A lazy implementation evaluates less parts of the ROBDD, i.e., less constructors if not the whole ROBDD is needed to compute the final result. For example if the implementation is completely lazy a function that yields the leftist path to a leaf causes the evaluation of the constructors on this path. The counting of the constructors uses about one percent of additional heap memory.

To get as meaningful results as possible we use a powerful PC for the measurements. The ROBDDs that are used in real live applications are very big and need a lot of memory and a fast processor for the processing. For the measurements we use a PC with 3GB of RAM and a 2GHz AMD Athlon XP 3000+ processor. We use GHC [14] version 6.4.1. For the BDD Package binding HBDD [12] we use GHC version 6.2.1. The HBDD binding is introduced in detail in Section 6.2. The package system that is used by HBDD is no more supported by the new GHC version. Unless otherwise noted we use no optimizations. The use of optimizations would improve the running time and decrease the memory consumption of the program but would make an interpretation of the results fairly complex. We would have to divide the results into effects of the optimizations and of the implementation. We execute the programs using the runtime system parameter `-H2G`. This option is called "suggested heap size" and tells the program to start with 2 gigabyte of heap memory. Without this parameter the incremental generational garbage collection that is used by the GHC would start with a small amount of heap memory and would increase the amount every time the heap memory is exhausted. The garbage collector is started every time the heap memory is exhausted. This would cause a lot of unnecessary garbage collector runs.

It is very difficult to choose good examples for the measurements. The algorithms on ROBDDs are still exponential in the number of variables in the worst case. In practical applications they tend to be much better. On the other side the ROBDDs that are used in practical applications are too big for the implementation and the test environment of this thesis. We cannot provide overall benchmarks for the implementations. We only look at some examples that are supposed to show a trend. We always compare two implementations, i.e., we state that one of them is better than the other in respect to a particular kind of expression. We try to give explanations for the differences to generalize the results.

3 ROBDD Implementations

This chapter presents three implementations of the ROBDD data structure. We start with an imperative-like implementation that is based on an implementation by Henrik Reif Andersen [3]. The second implementation is more functional. It uses an algebraic data type to define the structure of an ROBDD instead of a map structure. The third implementation focuses on the aspect of laziness and relaxes the *no-redundancy* property to gain any laziness in the construction of an ROBDD at all. We do not use destructive updates in all three implementations because our goal is a purely functional implementation.

3.1 An Imperative-like Implementation

Every node of an ROBDD is associated with a unique identifier. Its type is `NodeId` which is a type synonym for an integer type. The `NodeId` of a node uniquely determines the structure of the ROBDD that is rooted at this node. The unique Ids offer an efficient method to preserve the *no-redundancy* and the *sharing* property. To preserve the *no-redundancy* property we compare the `NodeId`s of the low and high successor when a node is constructed. If they are equal the node is not constructed because it is redundant. To preserve the *sharing* property we memoize all constructed nodes. We use a mapping from triples consisting of the `NodeId`s of the low and high successor and the variable number to the `NodeId` of the node. This way a new node with the same low and high successor and variable number gets the same `NodeId`. Because the construction works bottom-up this preserves the *sharing* property. In the imperative implementation this mapping is implemented by a dynamic hash table.

The structure of the ROBDD is defined by a second mapping. It maps the `NodeId` of a node to the `NodeId`s of the two successors and the variable number. In an imperative implementation this mapping is implemented by an array. This map represents the structure of the ROBDD and we refer to it as the *map*. Since the first map is the reverse mapping of this one we refer to it as the *reverse map*.

The ROBDD data type combines the *map* and the *reverse map*. The third argument of ROBDD is the `NodeId` of the root node. The terminals `Zero` and `One` are represented by the `NodeId`s zero and one respectively.

```
data ROBDD = ROBDD Map RevMap NodeId
```

In the imperative implementation the ROBDD is represented by the id of the root node while *map* and *reverse map* are global data structures.

3.1.1 Making a New Node

The first step to the construction of an ROBDD is the implementation of a function called `make`. This function adds one node to an ROBDD. It takes a variable number, the `NodeId`s of the low and high successor, the *map* and the *reverse map* and yields the resulting ROBDD. We call this function `rOBDD` because it is some kind of smart constructor for the ROBDD data structure. We assume the implementation of the abstract data types `Map` and `RevMap` that support lookup and insert functions. Table 3.1 shows these two ADTs. The abstract data type `RevMap` provides an additional function called `nextId` that yields the next free `NodeId`. The choice of a concrete implementation

```
insertMap :: NodeId → NodeId → Var → NodeId → Map → Map
lookupMap :: NodeId → Map → Maybe (NodeId, Var, NodeId)
insertRevMap :: NodeId → Var → NodeId → NodeId → RevMap
                → RevMap
lookupRevMap :: NodeId → Var → NodeId → RevMap → Maybe NodeId
nextId :: RevMap → NodeId
```

Table 3.1: The functions of the ADTs `Map` and `RevMap`

of these maps is discussed in Section 5.

The function `rOBDD` first checks whether the `NodeId`s of the two successors are equal. In this case it yields the ROBDD consisting of the unchanged *map* and *reverse map* and the `NodeId` of the low successor. This preserves the *no-redundancy* property because a node with two equal successors is never constructed. We do not have to change the maps because we do not construct a new node. If the `NodeId`s are not equal we look up whether a node with these successors and variable number already exists. If such a node is found an ROBDD with the unchanged map and *reverse map* and this `NodeId` is yielded. We do not have to change the maps because the node already exists in the *map* and in the *reverse map*. If the look-up fails the function `rOBDD2` is applied to the arguments.

```
rOBDD :: NodeId → Var → NodeId → Map → RevMap → ROBDD
rOBDD low var high map revmap
  | low==high = ROBDD map revmap low
  | otherwise =
    case lookupRevMap low var high revmap of
         Just nodeId → ROBDD map revmap nodeId
         Nothing     → rOBDD2 low var high map revmap
```

The function `rOBDD2` is a smart constructor that is not that smart. It simply adds the new node to the *map* and the *reverse map*.

```
rOBDD2 :: NodeId → Var → NodeId → Map → RevMap → ROBDD
rOBDD2 low var high map revmap =
  let nodeId = nextId revmap
  in
  ROBDD (insertMap nodeId low var high map)
        (insertRevMap low var high nodeId revmap) nodeId
```

3.1.2 Constructing an ROBDD

Based on the function `rOBDD` we define the function `build` that constructs an ROBDD out of a boolean expression. Later we replace this function by a construction that uses the function `apply` because the running time of `build` is exponential in the number of variables in the boolean expression.

The construction of an ROBDD is very similar to the construction of a Decision Tree. The difference is that we have to preserve the *no-redundancy* and the *sharing* property. This is achieved by using the smart constructor `rOBDD` instead of the constructor `DT`. The `build` function gets two additional arguments namely the *map* and the *reverse map*. These are passed from one application to another. To preserve the *sharing* property we have to construct the ROBDD in a certain order. That is, we have to decide to construct the ROBDD either from left to right or from right to left. We decide to do it left to right. This order is arbitrary but we have to keep it in mind for the definition of other functions later. It brings in a dependency of the low successors of a node on the high successors of a node. We pass the *map* and the *reverse map* that are yielded by the application of `build` on the low successor to the application to the high successor. The resulting *reverse map* is passed to `rOBDD`.

```
build :: BExp → ROBDD
build bexp = build' 1 bexp emptyMap emptyRevMap


build' :: Var → BExp → Map → RevMap → ROBDD
build' _ BFalse map revmap = ROBDD map revmap 0

build' _ BTrue map revmap = ROBDD map revmap 1

build' i bexp map revmap =
  let lowExp  = substitute bexp i BFalse
      highExp = substitute bexp i BTrue
      i' = succ i
      ROBDD lowMap lowRevmap low =
          build' i' lowExp map revmap
      ROBDD highMap highRevmap high =
          build' i' highExp lowMap lowRevmap
  in
  rOBDD low i high highMap highRevmap
```

Figure 3.1 illustrates the transfer of the *reverse maps* that are involved in the construction of a node. The `revmap` is passed to the construction of a node from the construction of the predecessor. This map is passed to the construction of the low successor which yields `lowRevmap`. This map is passed to the construction of the high successor which yields `highRevmap`. The `revmap` contains all predecessors and all nodes left of the con-

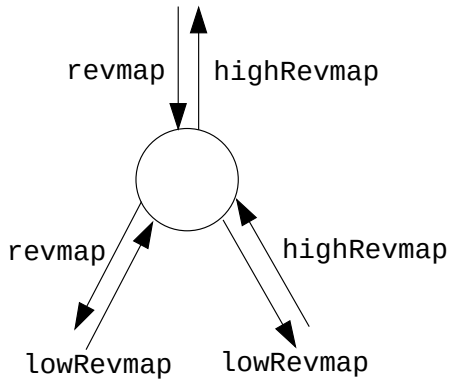

Figure 3.1: Transfer of the reverse map in the construction of a node

structed node. The `lowRevMap` additionally contains all nodes in the ROBDD rooted at the low successor. The `highRevmap` additionally contains all nodes in the ROBDD rooted at the high successor.

We do not have to look up the new node in the part of the *reverse map* that is added in the construction of its successors. Non of the nodes of the ROBDDs rooted at the two successors can be the same node as the one we are constructing. Therefore we do not have to look up the node in `lowRevmap` or in `highRevmap`. We look it up in `revmap`. This benefits the laziness because the constructed node is no longer dependent on all its successor nodes.

Figure 3.2 illustrates the difference between the two look-ups. The triangle represents the ROBDD. The dot marks the node that is constructed. A look-up in `highRevmap` depends on the horizontally and vertically lined parts of the ROBDD. A look-up in `revmap` depends only on the horizontally lined part. The higher the node, i.e., the smaller the variable number, the more the benefit. The extreme case is the construction of the root node. In this case `revmap` is the empty map while `highRevmap` contains all nodes of the ROBDDs rooted at the two successors of the root node, i.e., all nodes except for the root node. In this case the old implementation performs a look-up in a *reverse map* that contains all nodes except for one. The new implementation performs a look-up in the empty *reverse map*.

We still insert the new node to the *reverse map* that is passed to the predecessor namely `highRevmap`. The function `rOBDD` is enriched with an additional argument of type `RevMap`. We pass `revmap` and `highRevmap` to `rOBDD` and use them for the insert and look-up respectively.

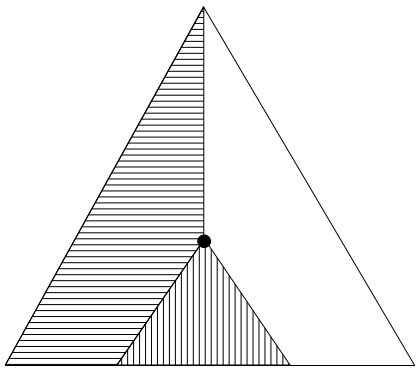

Figure 3.2: Dependencies in an ROBDD caused by look-ups in the *reverse map*

3.1.3 Descending an ROBDD

All operations on ROBDDs have to descend the ROBDD. We define the functions `getLow` and `getHigh` that yield the low and high successor of a node respectively. These functions make the implementation more flexible. For example for the implementation of complement edges like explained in Section 7.2 we mainly change the implementation of these two functions. Without these functions we would have to adapt all the pattern matchings of the operations on ROBDDs.

```
getLow :: NodeId → Map → NodeId
getLow nodeId map =
  case lookupMap nodeId map of
       Just (low,_,_) → low
       _              → error ("getLow:␣The␣node␣with␣NodeId␣"
                               ++ show nodeId
                               ++ "␣has␣no␣low␣successor")


getHigh :: NodeId → Map → NodeId
getHigh nodeId map =
  case lookupMap nodeId map of
       Just (_,_,high) → high
       _               → error ("getHigh:␣The␣node␣with␣NodeId␣"
                                ++ show nodeId
                                ++ "␣has␣no␣low␣successor")
```

The functions `getLow` and `getHigh` have the same complexity as `lookupMap`. Purely functional map implementations that support efficient look-up and insert functions are at least logarithmic in the number of elements in the map. For example the look-up and insert in a balanced search tree requires logarithmic time in the number of elements in the tree. The look-up and insert in a Braun [7] or Patricia Tree [26] are logarithmic in the key size. This is equal to the logarithm of the number of elements if the keys are continuous.

3.2 A More Functional Implementation

This section presents a more functional implementation of the ROBDD data structure. An ROBDD is a Decision Tree that satisfies two additional properties. We implement the ROBDD on the basis of the Decision Tree implementation in Section 1.3. The Decision Tree is implemented by an algebraic data type. The idea of the implementation of the ROBDD is to represent a directed acyclic graph by a tree with reference edges. These are edges that point at a node anywhere in the tree. We use the `NodeId`s of the ROBDD to point at a node. We use an algebraic data type similar to `DT` with an additional constructor for the reference edges.

We extend the algebraic data type that is used for the Decision Trees. We call the constructor `OBDD` instead of `DT` and add an argument of type `NodeId` to it. The leaves do not need a `NodeId` because `Zero` and `One` have the static `NodeId`s zero and one respectively. The reference edges are represented by the additional constructor `Ref`. We refer to a `Ref` constructor as a reference node and to an `OBDD` constructor as an original node.

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

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 in the `OBDD`. This decision is arbitrary but we have to remember it when we implement the consumer functions. A consumer function uses a preorder traversal. It uses the `NodeId` of the outermost `OBDD` constructor to memoize the results for all sub-`OBDD`s that are processed. We call a map that is used for memoization *memo map*. If the consumer function reaches a `Ref` constructor the result for this node is looked up by the `NodeId` of the `Ref`. Because the original node is the leftmost the consumer function visits the original node before it visits any references to this node. We have to assure that the original node is the leftmost to guarantee that equal sub-trees are processed only once by the consumer functions.

References that point at 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 memory of one unary constructor. The same holds for all `One` leaves. We do not memoize the application of a function to a leaf because the input is constant. Therefore the computation is not expensive. The memoizing of the result would be more expensive than the computation.

Figure 3.3 shows the `OBDD` data structure for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and a graphical representation of this `OBDD`. There is some indentation in the `OBDD` data structure to increase the readability. Constructors that have the same indentation are siblings. Arguments of a constructor are one level further right than the constructor itself.

In the graphical representation the constructors `Zero` and `One` are represented by the square nodes labeled 0 and 1 respectively. The `OBDD` constructors are represented by the

circle nodes. The labels of the nodes are the variable numbers. The `NodeId`s are left out. A `Ref` constructor is represented by an edge with a gap. The edge points at the node with the corresponding `NodeId`. The gap illustrates that we have no direct access to the node that is referenced. We only have access to its `NodeId`.

```
OBDD (OBDD Zero
           2
           (OBDD Zero
                 3
                 One
                 4)
           3)
     1
     (OBDD (Ref 4)
           2
           One
           5)
     2
```

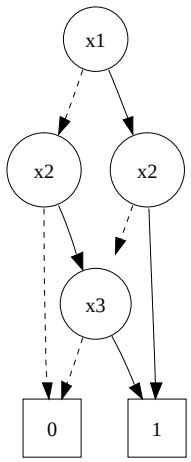

Figure 3.3: `OBDD` for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ (left) and a graph representation of this `OBDD` (right)

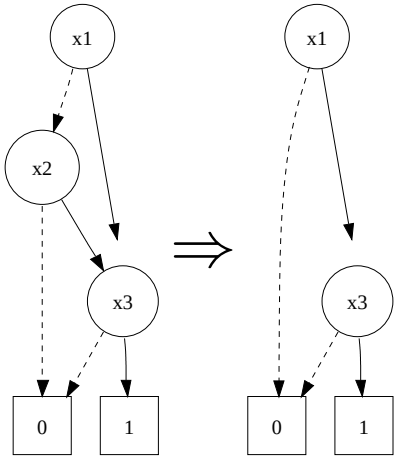

Figure 3.4: The application `restrict` x_2 `False` $((x_1 \wedge x_3) \vee (x_2 \wedge x_3))$ using `NodeId` references

The `restrict` operation chucks away parts of the `OBDD`. There are unpleasant cases in which it chucks away the original node but leaves a reference to this node. In this case we have to correct the `OBDD` data structure because we have to assure that the original node is the leftmost. We have to replace the leftmost reference node by the corresponding

original node. Figure 3.4 shows the `OBDD` for the expression $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and the result of restricting variable x_2 to `False` without an correction of the representation.

The original node that is labeled x_2 is chucked away but the reference to this node remains. We have to replace the reference `Ref 4` by the `OBDD` with the `NodeId` four. We would have to remember all sub-trees that are chucked away. The leftmost `Ref` to one of these nodes must be replaced by the original node. All other `Ref`s with the same `NodeId` remain. This requires a lot of bookkeeping. We would need a map that saves the nodes that are lost and have to look up every `NodeId` of a `Ref` constructor in this map.

The operation `apply` is a simultaneous traversal of two `ROBDD`s. It preserves the variable order by descending both root nodes if their variables are equal. If they are not equal only the node with the smaller variable is descended. Therefore there are cases in which `apply` demands the variable number of a `Ref` constructor. To get this number we have to look it up in an additional data structure or in the `OBDD` itself. This would require additional time and maybe memory. Section 4.1 presents the `apply` operation in detail. All the consumer functions do never demand the variable number of a reference node. They look up the `NodeId` in the *memo map*.

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

This new implementation solves the outlined problems. We replace the `NodeId`s in the reference edges by complete `OBDD`s. The operation `restrict` uses a *memo map* to memoize processed nodes. We have to look up the `NodeId`s of all references in this map. We do not save the `NodeId`s of nodes that are chucked away in this map. If the look-up of a reference fails we know that the original must be chucked away. To make the `OBDD` valid again we just remove the `Ref` constructor and apply `restrict` to this node. This adds its `NodeId` to the *memo map*. Therefore all other references with the same `NodeId` are not replaced.

Figure 3.5 shows the same restriction as Figure 3.4 but uses this new implementation. References are represented by an edge with an empty arrow head. To adjust the representation we just fill the arrow head, i.e., remove the outermost `Ref` constructor. Section 4.2 that presents the implementation of `restrict` explains this proceeding in more detail. The operation `apply` does not have to look up the variable number of a `Ref` constructor because it can directly access it.

We could omit the `Ref` constructors in the `OBDD` data type. That is, there would be no difference between a reference node and an original node in the `OBDD` representation. In this case all the functions that use memoization would have to look up the `NodeId`s of all nodes. Iff the look-up fails the node is an original node. In an implementation with `Ref` constructors all look-ups are successful. Note that this is not true for our `restrict` implementation. It is advantageous if all look-ups are successful because a look-up takes a logarithmic amount of time while checking whether a node is a `Ref` constructor takes constant time. Additionally it is advantageous for the laziness to look up as few `NodeId`s as possible. Every look-up causes the evaluation of some `NodeId`s in the map. Chapter 5

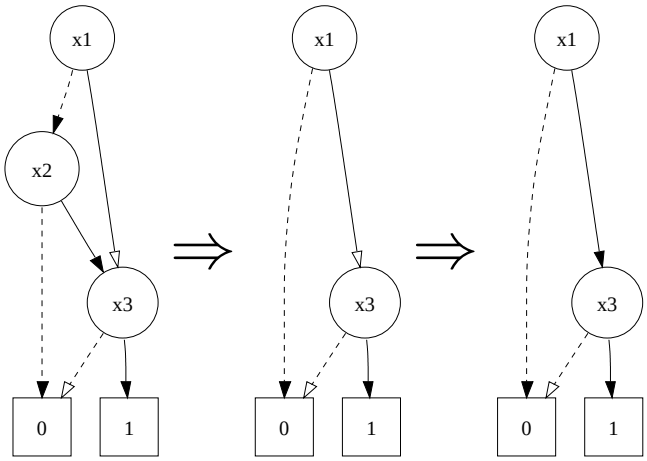

Figure 3.5: The application `restrict` x_2 `False` $((x_1 \wedge x_3) \vee (x_2 \wedge x_3))$ using `OBDD` references

that discusses the chose of a concrete implementation of the maps takes a closer look at this. Furthermore the information whether a node is a reference or not saves additional look-ups in the *reverse map* and look-ups by the `apply` operation. The prevention of these look-ups in the *reverse map* is essential for the laziness. On the other hand the additional `Ref` constructors require additional memory.

A function that uses memoization never evaluates an `OBDD` that is an argument of a `Ref` constructor. It only looks up its `NodeId` in the *memo map*. The `apply` operation does not satisfy this condition. There are cases in which `apply` causes the evaluation of both sub-trees, i.e., the original one and a reference to it and neither of them can be garbage collected. This may not happen because this would cause the same memory usage that is caused by a Decision Tree, i.e., an exponential amount of memory in respect to the number of variables. We additionally have to use implicit *sharing*, i.e., the *sharing* that is produced by the Haskell compiler. Lazy evaluation uses *sharing* to prevent that an expression is evaluated twice. For details see Section 1.1 about lazy evaluation or the semantics of lazy evaluation by John Launchbury [21]. Implicitly shared expressions are pointers to the same structure in the heap. This has too effects. First the expression is evaluated only once. Second all the shared structures only use the heap memory for one expression plus the memory for the pointers. We do not save the `NodeId`s of the nodes in the *reverse map* but the shared `OBDD` structure. When we look up a triple in the *reverse map* we place the shared `OBDD` structure that is yielded in a `Ref` constructor. We still need the explicit *sharing* that is provided by the `NodeId`s. Haskell provides no mechanism to check the pointer equality of two terms, i.e., to check whether two terms are shared. Therefore we need the `NodeId`s to identify equal sub-`OBDD`s.

The implicit *sharing* is disadvantageous for the laziness. The evaluation of an `OBDD` to Head Normal Form causes a look-up in the *reverse map*. If we would not use implicit *sharing* the look up would only be performed if the `NodeId` of the node is evaluated.

To save memory we merge every `Ref` constructor with the outermost `OBDD` constructor of its argument. We replace the `Ref` constructor by a `RefOBDD` constructor.

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

For the presentation in this paper we use a slightly different implementation. Instead of two distinct constructors we use an additional argument of type `Bool`. Iff its value is `True` the node is a reference, i.e., the whole `OBDD` that is rooted at this node is shared.

```
data OBDD = OBDD OBDD Int OBDD NodeId Bool
          | Zero
          | One
```

This implementation simplifies the definition of functions that make no use of the information whether a node is a reference or not. These functions have fewer rules. Note that this does not save running time since a pattern matching is translated into a jump and the running time is therefore independent of the number of rules. The implementation with the reference `Bool` requires more memory because every `OBDD` constructors gets an additional argument of type `Bool`.

In the `ROBDD` data structure we replace the `Map` by an `OBDD`. We remove the argument that holds the `NodeId` of the root node because the `OBDD` already provides this information.

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

3.2.1 Making a New Node

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

We only look up a node in the *reverse map* if its successors are both references. If one of its successors is not a reference it cannot be in the *reverse map*. The construction works bottom-up. If a node is in the *reverse map* all sub-trees of the tree that is rooted at this node are in the *reverse map*, too. Thus, if one sub-tree is no reference the node is none, too. If we construct a node that is no reference all its predecessors are no references. First of all this check saves running time because a look-up is more expensive than the check. Second this is essential for the laziness as is discussed in Section 3.3.1.

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

Again the function `rOBDD2` constructs the node. Instead of adding the node to the *map* we construct an `OBDD` with the two successors. This `OBDD` is added to the *reverse map*. The function `setRef` takes an `OBDD` and yields a corresponding reference, i.e., it replaces the boolean value of the outermost constructor by `True`. We deconstruct the outermost `OBDD` constructor and take a new one. This one is applied to the `NodeId`, low and high successor of the old constructor and to `True`. This function allocates the memory for the new `OBDD` constructor. The `setRef` function for the implementation that uses a `Ref` constructor instead of a reference boolean also allocates one constructor namely the `Ref` constructor.

All nodes in the *reverse map* are references. This way we do not have to apply `setRef` to the result of a look-up. We directly use the `OBDD` that is yielded by the look-up. That is, all reference nodes of an `OBDD` are implicitly shared while the original node only shares its successors with the corresponding reference nodes. Therefore n equal sub-OBDDs, where n is greater or equal 2, require the memory for the `OBDD` structure of the original node plus the memory for one constructor namely the constructor that is used by `setRef`. All the reference nodes are shared, i.e., they do not require additional memory. If all `OBDD`s in the *reverse map* would be no references we would need one additional `OBDD` constructor for every reference. That is, n equal sub-`OBDD`s would require the memory for one sub-`OBDD` plus the memory for $n - 1$ `OBDD` constructors.

3.2.2 Descending an ROBDD

The descending of an `OBDD` with this functional ROBDD implementation is more efficient than the descending of the imperative-like implementation.

```
getLow :: OBDD → OBDD
getLow (OBDD low _ _ _ _) = low

getLow obdd =
  error ("getLow:␣The␣OBDD␣" ++ show obdd
         ++ "␣has␣no␣low␣successor")


getHigh :: OBDD → OBDD
getHigh (OBDD _ _ high _ _) = high

getHigh obdd =
  error ("getHigh:␣The␣OBDD␣" ++ show obdd
         ++ "␣has␣no␣high␣successor")
```

The functions `getLow` and `getHigh` have logarithmic complexities in the imperative-like implementation. These two functions are in $O(1)$. Because of the implicit *sharing* even the descending of a reference node is in $O(1)$.

3.2.3 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 when it is applied to a Decision Tree it causes only the leftmost path to a `One` leaf and all parts left of it to be evaluated. Some parts of the `OBDD` will be additionally evaluated because of the *no-redundancy* and the *sharing* property.

The function `anySat` takes an ROBDD and yields a variable binding that satisfies this ROBDD if one exists. It uses a depth first strategy to find a `One` leaf.

```
anySat :: ROBDD → Maybe Binding
anySat (ROBDD robdd _) = anySatO robdd


anySat' :: OBDD → Maybe Binding
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)
       _              → error "anySat:␣ROBDD␣is␣not␣reduced"
```

This implementation of `anySat` yields the leftmost path to a `One` leaf. This is advantageous for the laziness because the original nodes are the leftmost and a reference node is only known to be a reference by looking it up in the *reverse map*. Therefore `anySat` tends to evaluate original nodes rather than reference nodes. In fact we show later that `anySat` only causes the evaluation of original nodes and performs no look-up in the *reverse map* at all.

There is no path for the `Zero` leaf and a path of length zero for the `One` leaf. The function yields a path for a node if one of its successors yields a path. There must be at least one path to a `One` leaf because otherwise the node is not reduced. If the application on the low successor yields a binding we add a binding of the current variable to `False` to it. Otherwise we check whether the other application is successful. The application `anySat' high` is not evaluated if `anySat' low` yields a path.

We apply the function `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 1.7. Figure 3.6 shows two observations made by Hood. The left one results from observing the `OBDD` data structure when applying `anySat` to the corresponding ROBDD. The right one shows the observation of a `DT` data structure for the same expression when applying `anySat`. In the observation of the `DT` data structure only a path to the leftmost `One` leaf and all parts left of it are evaluated. The `OBDD` data structure is almost completely evaluated. Although `anySat` does not pattern match against the `NodeId`s all ids except for the one of the root node are evaluated.

The pattern matching of `anySat` forces `build'` to evaluate the `OBDD` to Head Normal

```
Just [(1,False),(2,True),(3,True)]
```

```
>>>>>>> Observations <<<<<<
testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    4
                    False)
              3
              False)
        1
        (OBDD (OBDD Zero
                    3
                    One
                    4
                    True)
              _
              One
              5
              False)
        _
        False)
```

```
>>>>>>> Observations <<<<<<
testExp4
  (DT (DT Zero
          2
          (DT Zero
              3
              One
              _)
          _)
      1
      _
      _)
```

Figure 3.6: Observations of an `ROBDD` (left) and a `DT` (right) when applying `anySat`

Form. Because the boolean expression is neither `BFalse` nor `BTrue` the function `rOBDD` is demanded to yield the Head Normal Form of the `OBDD`. `rOBDD` has to check which guard is satisfied. To check the first guard it demands the `NodeId`s of the two successors. The function `getId` demands the successors to be evaluated to Head Normal Form, i.e., demands `build'` to yield their Head Normal Forms. This forms the vicious circle. This is highly simplified but points out the problem. To evaluate any `OBDD` to Head Normal Form we have to evaluate the `NodeId`s of its two successors. This is caused by the *no-redundancy* property. The evaluation of these `NodeId`s again causes the evaluation of the `NodeId`s of their successors and so on. This results in the complete evaluation of the `OBDD` data structure.

If we want to check whether a node is redundant we have to compare its successors somehow. We have to evaluate the successors at least as far as they are equal. This definitely causes the evaluation of the outermost constructors of the successors. If we check the *no-redundancy* property for all nodes this causes the whole structure to be evaluated. The only way to avoid this is to relax the *no-redundancy* property for some

nodes.

Although an ROBDD with relaxed *no-redundancy* property is no ROBDD 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. The next chapter examines an ROBDD implementation with relaxed *no-redundancy*.

3.3 A Lazy Implementation

This chapter presents an implementation of the construction of an ROBDD with relaxed *no-redundancy* property and discusses the consequences of the relaxing.

3.3.1 Lazy Nodes

We call all nodes that are known to be no reference without looking them up in the *reverse map lazy* nodes. By a node with two leaves we denote a node whose low successor is the `Zero` leaf and high successor is the `One` leaf or vice versa. In an ROBDD all nodes on the path to the leftmost node with two leaves and left of it are *lazy* nodes. The leftmost node with two leaves is the first node that is looked up in the *reverse map*. This node is guaranteed to be not in the *reverse map* because the map is empty. Therefore this node is no reference. That is, all predecessors of the leftmost node with two leaves are not looked up, too. We do not look them up because we only look up nodes whose successors are both references.

The number of nodes on the path to the leftmost node with two leaves and left of it is in $O(n)$ where n is the number of variables. There are two cases. Either the low successor of a node of this path belongs to the path itself or it is a leaf. If the low successor does not belong to the path the high successor has to. We assume that the low successor is no leaf. Then the ROBDD that is rooted at this node must contain a node with two leaves. Otherwise it would be reduced to a single leaf because of the *no-redundancy* property. Therefore the high successor would not be part of the path to the leftmost node with two leaves. This is a contradiction. That is, all nodes in an ROBDD that are left of the leftmost path to a node with two leaves are leaves. The path consists of at most n nodes and there are at most n leaves left of this path. Thus the number of nodes on the path to the leftmost node with two leaves and left of it is in $O(n)$. Because all parts that are left of the path to the leftmost node with two leaves are leaves we do not explicitly refer to these. That is, if we talk about the evaluation of the path to the leftmost node with two leaves the leaves left of this path are always included.

If a node is not *lazy* all the nodes of the ROBDDs that are rooted at the two successors of this node are not *lazy*, too. If any of these nodes would be *lazy* it would be no reference node. That is, we would not have to look up its successor. By iterated use of this rule we would not have to look up the node we started with. That is, this node would be *lazy* which is a contradiction.

The *lazy* nodes are essential for the laziness. To look up a node in the *reverse map* we have to evaluate the `NodeId`s of its successors. Without *lazy* nodes the whole ROBDD would be evaluated even without the *no-redundancy* property because all the `NodeId`s are evaluated by the look-ups. We do not relax the *no-redundancy* property for nodes that are looked up in the *reverse map*. The `NodeId`s of the two successors of these nodes are evaluated anyway because of the look-up. We relax the *no-redundancy* property for the *lazy* nodes, i.e., the nodes that have at least one successor that is no reference.

3.3.2 Relaxing the No-Redundancy Property

An ROBDD with relaxed *no-redundancy* property has more nodes than an ROBDD with full *no-redundancy* property. This worsens the running times of the operations on this ROBDD. Besides this an ROBDD with relaxed *no-redundancy* property is no more canonical. That is, there are more than one ROBDD with relaxed *no-redundancy* that represent the same boolean function. We can add redundant nodes to the ROBDD and the represented boolean function stays the same. That is, there are as many ROBDDs that represent the same boolean function as redundant nodes can be added.

Figure 3.7 shows the three cases in which a node can be removed because of the *no-redundancy* property. An implementation that checks the *no-redundancy* property only for *lazy* nodes does not remove a node where the high successor is a reference to the low successor as seen in the left graphic. The node is removed if both successors are references to the same node. There are no nodes in the ROBDD where the low successor is a reference to the high successor because the original node is always the leftmost node in the `OBDD`.

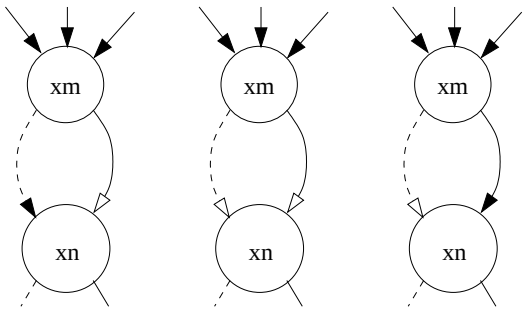

Figure 3.7: The three types of redundant nodes

Let f be an ROBDD with full *no-redundancy* property. Let e be an edge in f that leads from a node with variable u to a node with variable v. In an ROBDD with relaxed *no-redundancy* property there can be $u - v - 1$ redundant nodes on edge e, i.e., one redundant node for every variable between the variables of the connected nodes. Because we keep the *no-redundancy* property for the leaves we get no additional nodes for edges that lead to a leaf. We cannot use this insight for the estimation because we do not know how many edges lead to a leaf. We can roughly estimate the number of redundant nodes on one edge by the number of variables minus one, i.e., $n - 1$.

If a node is a reference both its successors are references. We check the *no-redundancy* property for nodes whose successors are both references. Therefore we remove all redundant nodes on edges that lead to a reference. Only edges that lead to original nodes can contain redundant nodes. The number of edges that lead to an original node is equal to the number of nodes in the ROBDD. We can estimate the number of redundant nodes by the upper bound of $(n-1) * |f|$ where $|f|$ is the number of nodes in the ROBDD with full *no-redundancy* property.

Redundant nodes can destroy the *sharing* of other nodes. Figure 3.8 shows two ROBDDs that represent the same boolean function. The left ROBDD contains no redundant nodes. We add one redundant node that checks the variable x_3. We check this variable only in the case that variable x_1 is *false*. First this destroys the *sharing* of the two ROBDDs rooted at the nodes labeled x_2. Because these sub-ROBDDs are no more equal the node labeled x_1 is no more redundant. The ROBDD where we have added one redundant node has seven nodes while the ROBDD with full *no-redundancy* has only four nodes. That is, we cause a blow up by three nodes with one redundant node.

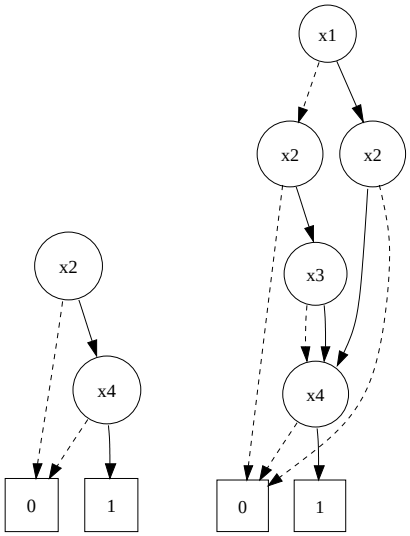

Figure 3.8: ROBDDs for the expression $x_2 \wedge x_4$ with full *no-redundancy* property (left) and relaxed *no-redundancy* property (right)

A redundant node cannot destroy the *sharing* of a whole sub-tree. The successors of the redundant node are still shared. Only the sharing of the predecessors of a redundant node can be destroyed. That is, a redundant node can destroy the *sharing* of the path from the root node to the redundant node. All other nodes are still shared. In the example the redundant node is the node labeled x_3. The successor of the redundant node, the node labeled x_4, is still shared. The *sharing* of the predecessor of the redundant node namely the left node labeled x_2 is destroyed. Additional nodes are caused by the redundant node only on a path from the root node to the redundant node.

Because we can only destroy the *sharing* of the nodes on the path from the root node to another node we can cause at most n additional nodes by the insert of one redundant

node. There are at most $n-1$ redundant nodes per node in an ROBDD. Therefore an ROBDD with relaxed *no-redundancy* property can have at most $n^2|f|$ nodes. This is a very loose estimation. We would need more time to deliver a closer estimation of the number of nodes in an ROBDD with relaxed *no-redundancy* property. The same holds for the check whether the destroying of the *sharing* can be prevented by the implementation. The measurements in this paper show that the number of additional nodes that are caused by the relaxing is small for all the examples.

If we relax the *no-redundancy* property the ROBDD representation is no more canonical. We do not relax the *no-redundancy* property for the leaves because in this case the complexity of the satisfiability and the tautology check would be in $O(n)$ while for ROBDDs with full *no-redundancy* property they are in $O(1)$. Additionally we gain no extra laziness if we relax the *no-redundancy* property for the leaves. This is discussed in Section 3.3.3. We relax the *no-redundancy* property only for inner nodes.

For a canonical representation the equality check is a check for isomorphy. This can be done in $O(min\{|f|,|g|\})$ where $|f|$ and $|g|$ denote the number of nodes of the two ROBDDs. The equality check for an ROBDD with relaxed *no-redundancy* property is more difficult. We can reduce the ROBDD with the full *no-redundancy* property and use the isomorphy check on the results. The running time of this implementation will be worse than the running time of the equality check for ROBDDs with full *no-redundancy* property. Besides the additional nodes this is the price that we have to pay for the laziness.

Surprisingly there is an equality check implementation for ROBDDs with relaxed *no-redundancy* that is almost as good as the check for ROBDDs with full *no-redundancy*. This implementation makes use of the laziness and is better than the check for ROBDDs with full *no-redundancy* if two unequal ROBDDs are compared. If two equal ROBDDs are compared the implementation with relaxed *no-redundancy* is worse than the implementation with full *no-redundancy* but the differences are small. The different implementations of the equality check are discussed in detail in Section 4.3.

3.3.3 Making a New Node

We pull the equality check of the `NodeIds` in the function `rOBDD` to the inside of the function. We check the *no-redundancy* property only for nodes that are not *lazy*, i.e., nodes whose successors are both references. We could remove the check for redundancy completely. Its benefit for the laziness depends on the laziness of `lookupRevMap`. If it can yield `Nothing` by evaluating only one of the keys, i.e., the low successor, high successor or variable number, a complete removal would be advantageous for the laziness. There are map implementations that provide this property. The probability that `lookupRevMap` yields `Nothing` without evaluating all the keys decreases with every inserted node. Therefore the benefit of a complete removal is negligible as we have checked with some measurements.

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

rOBDD One _ One _ revmap = ROBDD One revmap

rOBDD low var high revmap1 revmap2
  | isRef low && isRef high =
    if getId low==getId high
       then ROBDD low revmap1
       else
        case lookupRevMap low var high revmap2 of
          Just obdd → ROBDD obdd revmap1
          Nothing   → rOBDD2 low var high revmap1
  | otherwise = rOBDD2 low var high revmap1
```

The function `getId` yields the `NodeId` of the root node, i.e., the `NodeId` of the outermost `OBDD` constructor. We do not compare `low` and `high` but `getId low` and `getId high`. It would be a bad semantics of equality if the two `OBDD`s are equal iff the `NodeId`s of the root nodes are equal. This is only true if we would use the full *no-redundancy* property.

The laziness does not benefit from a relaxing of the *no-redundancy* property for the leaves. If we relax the *no-redundancy* property for the leaves `rOBDD` causes the same evaluation as with the check for *no-redundancy* for leaves. The rules of `rOBDD` are translated into a cascade of case expressions. The following code illustrates the result of the translation. The expression `exp` represents arbitrary expressions that are of no relevance for the example.

```
rOBDD low var high revmap1 revmap2 =
  case low of
       Zero → case high of
                   Zero → exp
                   _    → exp
       One  → case high of
                   One  → exp
                   _    → exp
       _    → exp
```

The outermost case matches the first argument of `rOBDD` against `Zero` and `One`. This pattern matching evaluates the first argument to Head Normal Form. If one of the patterns matches, i.e., if the first argument is a leaf, the second argument is evaluated to Head Normal Form. If none of these patterns matches, i.e., if the first argument is a `OBDD` constructor, the second argument is not evaluated.

If we relax the *no-redundancy* for leaves we remove the first two rules of `rOBDD`. That is, there is no pattern matching against the arguments of `rOBDD`. The check whether both successors are references still causes the same evaluation as the pattern matching. The boolean conjunction `(&&)` is lazy, i.e., it only evaluates its second argument if the evaluation of the first argument yields `True`. If the first argument is a leaf it is defined to be a reference. Therefore the second argument is evaluated to Head Normal Form.

If leaves would be no references we would never share any nodes. The second argument is evaluated to Head Normal Form iff the first is a leaf. Thus, without *no-redundancy* property for the leaves the same parts of the arguments of `rOBDD` are evaluated as with the *no-redundancy* property. A measurement proved that there is no difference in the number of evaluated constructors with or without *no-redundancy* property for leaves.

3.3.4 Laziness

We build 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 3.3.4 shows the observations of the implementation with relaxed *no-redundancy* property. The right part shows the observations for the ROBDD implementation with full *no-redundancy* property. The whole high successor of the root node is not evaluated

```
Just [(1,False),(2,True),(3,True)]
```

```
>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    _
                    False)
              _
              False)
        1
        _
        _
        False)
```

```
>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    4
                    False)
              3
              False)
        1
        (OBDD (OBDD Zero
                    3
                    One
                    4
                    True)
              _
              One
              5
              False)
        _
        False)
```

by the implementation with relaxed *no-redundancy* while it is by the implementation with full *no-redundancy*. In the example with relaxed no-redundancy property all evaluated

nodes are *lazy* nodes. That is, we do not look them up in the *reverse map*. All these nodes are known to be *lazy* nodes because their low successors are original nodes. Therefore the boolean values that state whether a node is a reference are evaluated but the `NodeId`s are not.

Evaluation to Head Normal Form

The *no-redundancy* property causes the evaluation of the whole `OBDD` data structure if the `OBDD` is evaluated to Head Normal Form (HNF). Even the relaxed *no-redundancy* property works against the laziness. We investigate which parts of an `OBDD` are evaluated if we evaluate it to HNF. We distinguish whether the root node of the `OBDD` that is evaluated to HNF is *lazy* or not.

We take a look at an examples. We construct the ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. We observe the `OBDD` data structure when the function `getVar` is applied to this ROBDD. The left part of Figure 3.9 shows this observation. The function `getVar` yields the variable of the outermost `OBDD` constructor. It has to check whether the `OBDD` is a leaf, i.e., it has to evaluate the `OBDD` to HNF. The root node of the `OBDD` that is evaluated to HNF is a *lazy* node. Therefore its low successor namely the `OBDD` rooted at variable `2` is evaluated to HNF. This causes the evaluation of its low successor to HNF. The low successor of this node is a `Zero` leaf. The function `rOBDD` checks whether the high successor is a `Zero` leaf, too. This causes the evaluation of the `OBDD` rooted at variable `3` to HNF. Its low successor is a `Zero` leaf. Therefore the high successor is evaluated to HNF, too. This one is a `One` leaf. This node is not redundant, i.e., it is an `OBDD` constructor. Therefore all predecessors of this node are `OBDD` constructors, too. That is, the outermost constructor is known to be an `OBDD` constructor.

The definition that follows is recursive. That is, the evaluation of an `OBDD` to HNF causes the evaluation of sub-`OBDD`s to HNF. The evaluations of the sub-`OBDD`s follow the same rules as the evaluation of the top level `OBDD`. We describe the evaluation of an `OBDD` to HNF by means of the evaluation of sub-`OBDD`s to HNF.

If the root node of the `OBDD` that is evaluated to HNF is a *lazy* node we have to check whether it can be reduced to a leaf. That is, we have to check whether both successors are the same kind of leaf. This causes the evaluation of the low successor to HNF. If the low successor is a leaf the high successor is evaluated to HNF, too. A node with two leaves, i.e., low successor `Zero` and high successor `One` or the other way round is not redundant. If such a node is evaluated all predecessors of this node are known to be `OBDD` constructors because at least one of their successors is no leaf.

If a top level `OBDD`, i.e., an argument of an `ROBDD` constructor is evaluated to HNF a path to the leftmost node with two leaves is evaluated. If the low successor of a node is a leaf the high successor is evaluated to HNF. If the low successor is the other kind of leaf the node is known to be an `OBDD` constructor and all predecessors, too. That is, as soon as a node with two leaves is evaluated the root node is known to be an `OBDD` constructor. Because Haskell evaluates function arguments from left to right the function `rOBDD` causes the evaluation of the low successor of a node to HNF before the high successor is evaluated. According to Section 3.3.1 the nodes on the path to the

```
1

>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    _
                    False)
              _
              False)
        1
        _
        _
        False)
```

```
2

>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    2
                    False)
              _
              False)
        1
        (OBDD (OBDD Zero
                    3
                    One
                    2
                    True)
              2
              One
              _
              False)
        _
        False)
```

Figure 3.9: Observations of two evaluations to Head Normal Form

leftmost node with two leaves are *lazy* nodes. Therefore the evaluation of a top level `OBDD` to HNF causes the evaluation of sub-`OBDD`s that are rooted at *lazy* nodes.

In the left observation in Figure 3.9 the root node of the `OBDD` that is evaluated to HNF is a lazy node. According to the previous paragraph this causes the evaluation of the path to the leftmost node with two leaves. Note that this includes all the leaves left of this path like stated in Section 3.3.1. In the observation exactly these parts are evaluated.

The `OBDD` for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ is observed when the function `getVar` is applied to the high successor of this `OBDD`. The right part of Figure 3.9 shows these observations. The function `getHigh` causes the evaluation of the `OBDD` to HNF. This causes the same evaluation as the application of `getVar` in the left part of Figure 3.9. The application of `getVar` to the high successor additionally causes the evaluation to HNF of the `OBDD` that is rooted at the high successor.

The root node of this `OBDD` is not *lazy* because its low successor is a `Zero` leaf and its high successor is a reference. Therefore the whole `OBDD` rooted at this node is evaluated.

The low successor of this node is a reference. The evaluation of this `OBDD` causes the evaluation of the `OBDD` that is referenced. This `OBDD` is positioned in the low successor of the root node. This `OBDD` would not be evaluated without the evaluation of the reference.

The `NodeId` of a node that is not *lazy* depends on the structure of the whole `OBDD`. Therefore the evaluation of such a `NodeId` causes the evaluation of the whole `OBDD`. Note that the evaluation of the `NodeId` of a *lazy* node does not cause the evaluation of the whole `OBDD`.

If the root node of the `OBDD` that is evaluated to HNF is not *lazy* we have to check whether the node is redundant. The outermost constructor depends on the result of the comparison of the `NodeId`s of the successors. This comparison causes the evaluation of the whole `OBDD`.

An application of `anySat` to an ROBDD checks for every node on the path to the leftmost `One` leaf whether it is an `OBDD` constructor or not. This causes the evaluation to HNF of all the `OBDD`s rooted at these nodes. According to Section 3.3.1 the nodes on the path to the leftmost node with two leaves and left of it are all *lazy*. The path to the leftmost `One` leaf is part of these nodes. Therefore all root nodes of the `OBDD`s that are evaluated to HNF are *lazy* nodes. An application of `anySat` to an ROBDD causes the evaluation of the path to the leftmost node with two leaves and all parts left of it. Therefore `anySat` evaluates $O(n)$ many nodes where n is the number of variables in the ROBDD. No look-ups or inserts in the *reverse map* are performed by `anySat` because all the evaluated nodes are *lazy*. That is not exactly true because the node with two leaves is looked up in the *reverse map*. Because the map is empty the look-up yields `Nothing` without evaluating the empty map. This is addressed in detail in Section 5 where we discuss several implementations of the *reverse map*.

Assigning NodeIds

We number the nodes of an ROBDD in postorder. That is, we first number the low successor then the high successor and in the end the node itself. This is the canonical numbering because the look-up in the *reverse map* depends on the `NodeId`s of the successors. That is, we have to number the successors before we can number the node. We could as well use a reversed preorder, i.e., first number the high successor then the low successor and the node in the end. This would be disadvantageous because it would bring in a dependency of the `NodeId`s of the low successor on the ones of the high successor while the dependency that is brought in by the construction order is reversed.

If a postorder numbering is used the `NodeId` of the root node depends on the greatest `NodeId` of the high successor. Therefore we have to evaluate this `NodeId` to evaluate the `NodeId` of the root node. This `NodeId` again depends on other `NodeId`s. If one of the nodes whose `NodeId`s are evaluated is not *lazy* the evaluation of the `NodeId` causes the evaluation of the whole `OBDD`. To break these dependencies we have to use gaps in the numbering. We do not know whether the constructed node gets a new `NodeId` until we have looked up the node in the *reverse map*. That is, we have to save a `NodeId` for the node and do not use this `NodeId` in the numbering of its successors. If the node turns out to be a reference we do not use the saved `NodeId` at all. This causes a gap in

the numbering because one number is left out. If the node is no reference we just use the `NodeId` we have saved before. This way we can use a preorder numbering. With a preorder numbering the `NodeId` of the node does not depend on the `NodeId`s of the nodes of the `OBDD`s rooted at the two successors.

There still remain dependencies in the numbering when we use a preorder numbering. That is, to number the high successor we have to know the greatest `NodeId` in the low successor. These dependencies cause no additional nodes to be evaluated. The dependencies that are caused by the preorder numbering are the same that result from the look-up in the *reverse map*. Figure 3.2 illustrates these dependencies. With the preorder numbering the `NodeId` does not depend on the `NodeId`s of its successors but on the `NodeIds` of the parts of the `OBDD` left of it namely the vertically lined parts in the graphic.

Instead of the preorder we could use a static numbering like the one used in Braun Trees. This numbering does not have any dependencies at all. When we use a Braun numbering the `NodeId`s grow exponential in the number of variables. Therefore we have to use the type `Integer` for the `NodeId`s instead of `Int`. The values of the type `Integer` are unbounded integer values. Otherwise the number of variables would be limited to 64 because there are 2^{64} `Int` values.

Figure 3.10 shows the number of evaluated constructors for a postorder, a preorder and a Braun numbering. In three of the examples the preorder and Braun numbering causes the evaluation of fewer constructors than the postorder numbering. In all the other cases there is no difference. The preorder and Braun numbering always cause the evaluation of the same number of constructors, i.e., there is no advantage of the static numbering over the preorder numbering. On the other hand the `NodeId`s that are used by the preorder numbering are much smaller than the ones of the Braun numbering.

A numbering with gaps can increase the expense of a look-up and insert in the maps. For example a look-up or insert in a Braun or Patricia Tree takes logarithmic time in the size of the key. A numbering with gaps increases the size of the `NodeId`s, that is the size of the keys. Therefore a numbering with gaps increases the time and memory consumption if we use a Braun or Patricia Tree for the implementation of the maps. The measurements in Section 5 advise to use a Braun Tree implementation for the maps. We have investigates whether the benefit of lesser evaluated constructors does exceed the disadvantages of the gaps in the numbering. In the examples were less constructors are evaluated the time and memory consumption of the Braun Tree implementation benefits. In the cases where the same number of constructors are evaluated the preorder numbering is a disadvantage because the look-ups and inserts are more expensive. That is, both implementations have their advantages and disadvantages. Therefore we use a Braun Tree implementation with the more conservative postorder numbering of the `NodeId`s in all measurements in this paper.

As mentioned before the function `build` is exponential in the number of variables in the expression. We do not use this function for the construction of ROBDDs. Although the results stated here can be applied to other operations. The `build` operation is a combination of a construction of a `DT` and a reduction to an `ROBDD`. This scheme is used in all operations that yield an `ROBDD`. Instead of doing this one after the other it is done

Expression	Numbering	Eval. Constr.
Queens 7	Postorder	598430
	Preorder	497926
	Braun	497926
Integer 17	Postorder	236
	Preorder	236
	Braun	236
Integer2 1000	Postorder	505498
	Preorder	505498
	Braun	505498
HWB 14	Postorder	2044822
	Preorder	2044822
	Braun	2044822
uf20-02.cnf	Postorder	4689
	Preorder	4324
	Braun	4324
uf20-020.cnf	Postorder	27078
	Preorder	21835
	Braun	21835

Figure 3.10: Number of evaluated constructors for several numberings

all at once. This saves time and memory. The results are applicable to all operations of this kind because the construction of the `DT` does not bring in any dependencies. The additional evaluations are all caused by the reduction. The only additional evaluation that is caused by the construction of the `DT` is the evaluation of the variable numbers. These are evaluated because of the substitution in the boolean expression. We can apply these observations to all operations that yield an ROBDD and therefore reduce an `DT`. This is also the case in the function `apply` that is later used to construct an `ROBDD`. Note that this results does not say anything about which parts of the arguments of `apply` are evaluated. It only concerns the resulting `ROBDD`.

4 Implementation of a BDD Package

This chapter presents the implementation of the main operations for the ROBDD implementation with relaxed *no-redundancy* property.

4.1 Apply

Every application of `build` causes two recursive applications. Every application removes one variable number from the expression. Therefore `build` is exponential in the number of variables.

The `apply` operation combines two ROBDDs with a boolean operator. The imperative implementation of this function has a worst case running time in $O(|f||g|)$ where f and g are the argument ROBDDs. We can construct an ROBDD by replacing every operator in a boolean expression by an appropriate application of `apply`. The ROBDDs for a single variable and the constants *true* and *false* are easy to construct. This construction is also exponential in the worst case. This case occurs when an ROBDD has exponential many nodes in the number of variables.

For the construction of an `ROBDD` out of a boolean expression it is sufficient to provide a negation and an `apply` function for the boolean operator $\vee$. For example an `apply` with the boolean operator $\Leftrightarrow$ can be expressed by three applications of `apply` and five applications of `negate`. One of these `apply` applications is applied to ROBDDs of the same size as the application `apply (⇔)`. The others are applied to smaller ROBDDs. Therefore the implementation using the boolean operator $\Leftrightarrow$ is more efficient than the combination of `apply`s. Thus we define an `apply` operation that takes an arbitrary boolean operator as first argument. This leaves the choice to the user whether to use one `apply` with an appropriate boolean operator or an equivalent combination of `apply` operations. With the power of higher order functions like provided by Haskell this is fairly easy. The user interface that is presented in Section 4.5 provides a simple mechanism to alter the implementation of the provided boolean operators. All boolean operators provide a standard implementation that uses a combination of `apply`s. By defining one of these boolean operators this standard implementation is replaced.

The implementation of `apply` is based on the observation that we can push all boolean operators down to the arguments of a *Shannon Expansion*.

$$apply\ op\ f\ g = x \rightarrow \begin{array}{l} apply\ op\ (f[x \mapsto 0])\ (g[x \mapsto 0]), \\ apply\ op\ (f[x \mapsto 1])\ (g[x \mapsto 1]) \end{array}$$

The application of an operator *op* to the ROBDDs f and g can be expressed by a *Shannon Expansion* of a variable x and the substitution of this variable in the ROBDDs f and g by *false* and *true* respectively.

Like `build` the function `apply` has to choose a top variable for the *Shannon Expansion*, i.e., the x in the expansion above. We define a `substitute` function on ROBDDs. This function assumes that the variable that is substituted (`var`) is less equal to the top variable of the ROBDD. In this case the implementation of the `substitute` function is very simple. The function `topVar` yields the variable of the root node of an ROBDD.

```
substitute robdd x b
  | x<topVar robdd = robdd
  | x=topVar robdd = if b
                       then getHigh robdd
                       else getLow robdd
```

If `x` is less than the top variable we know that it does not occur in `robdd` at all because the variables in the ROBDD are ordered. If the variables are equal we just yield the low or high successor dependent on whether we substitute `x` by `False` or `True`.

We are looking for a variable that is less equal to both top variables of the argument ROBDDs. If we substitute such a variable the substitutions that are used in the expansion are very simple. We use the smaller variable of the two top variables. The function `next` chooses the smaller of the two variables and applies the substitutions to the ROBDDs. It yields the four ROBDDs, namely $f[x \mapsto 0]$, $g[x \mapsto 0]$, $f[x \mapsto 1]$, $g[x \mapsto 1]$ and the top variable x. This operation is in $O(1)$.

The terminal case of the application of `apply` is the application to two leaves. In this case the boolean operator is applied to the corresponding boolean values.

Figure 4.1 shows the ROBDDs for the expressions $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$ and the result of applying $\wedge$ to these ROBDDs. The nodes are labeled with their `NodeId`s. The variable numbers of the nodes are shown on the left. The `NodeId`s are needed for the callgraph. Figure 4.2 shows the callgraph for this application of `apply`. Each node represents the application of `apply` to two ROBDDs. The nodes are labeled with the `NodeId`s of the root nodes of these ROBDDs.

Memoization is used to apply `apply` only once to a pair of ROBDDs. For every pair of `NodeId`s the result of the application is saved in a map called *apply map*. The dashed nodes represent applications that are memoized and therefore looked up in the *apply map*. Applications to two leaves are not memoized. The input of this applications is constant and the insert and look-up are more expensive than the computation.

Some parts of the callgraph reduce to `Zero` leaves. An example is the sub-tree rooted at the node labeled $4, 0$. The results of the applications of $\wedge$ to the leaves of this sub-callgraph are `Zero` leaves. Therefore this whole sub-tree is reduced to a single leaf. The semantics of the node labeled $4, 0$ is the application of the boolean operator $\wedge$ to a boolean expression and *false*. This yields *false* independent of the first argument. In this case we do not have to traverse the ROBDD. We can immediately yield the `Zero` leaf. The same holds for applications of `apply` with the boolean operator $\wedge$ where the first argument is the `Zero` leaf. All bold nodes in Figure 4.2 represent applications that immediately yield the `Zero` leaf without traversing one of the arguments. This idea was introduced by Bryant [8]. Let a and b be boolean values. If a boolean operator *op* satisfies $a\ op\ false = a\ op\ true = b$ we can define $a\ op\ X = b$ where X is a *don't care*

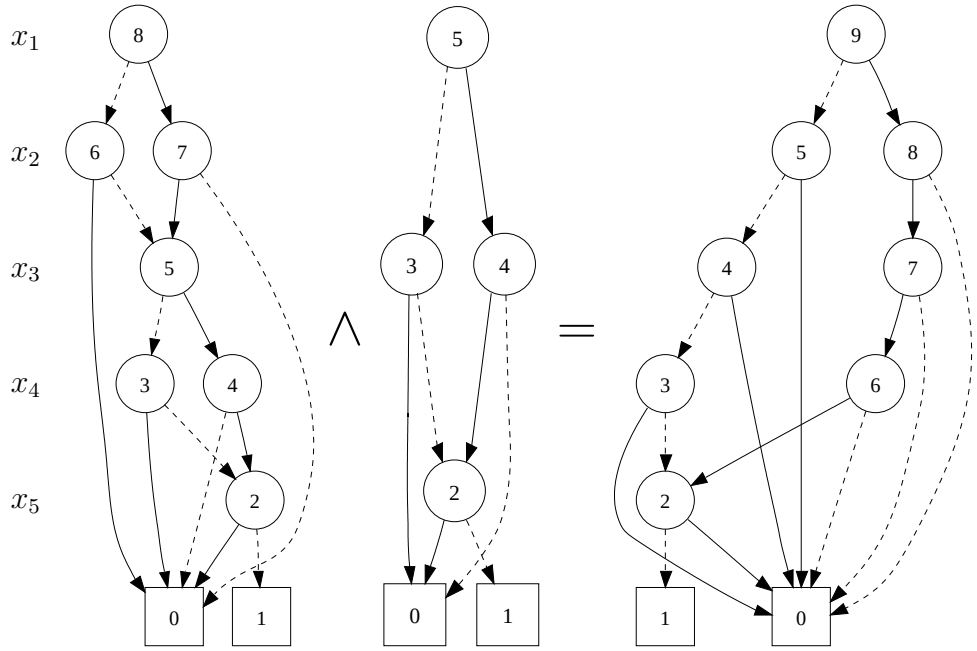

Figure 4.1: ROBDDs for the expressions $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$ and the result of applying $\wedge$

value. That is, no matter which value the second argument has we know the result.

The first argument of `apply` is the boolean operator that is applied to the leaves. Due to the *don't care* idea its type is `Maybe Bool -> Maybe Bool -> Maybe Bool`. This operator is a function on a three valued logic. The values `Just True` and `Just False` represent the two boolean values. The value `Nothing` represents a third value that is neither *true* nor *false*, i.e., a boolean expression that cannot be simplified to one of the boolean values. The function `obddToBool` takes an `OBDD` and yields the corresponding value of the three value data type. That is, it yields a boolean value for the leaves and `Nothing` otherwise.

The function `andM` implements the boolean operator $\wedge$. It yields `Just False` if one of the arguments is `Just False`. On the Haskell level *don't care* values are represented by underscores. That is, we do not need the value of this argument to express the result.

```
andM :: Maybe Bool → Maybe Bool → Maybe Bool
andM (Just False) _ = Just False

andM (Just True) x = x

andM _ (Just False) = Just False

andM _ _ = Nothing
```

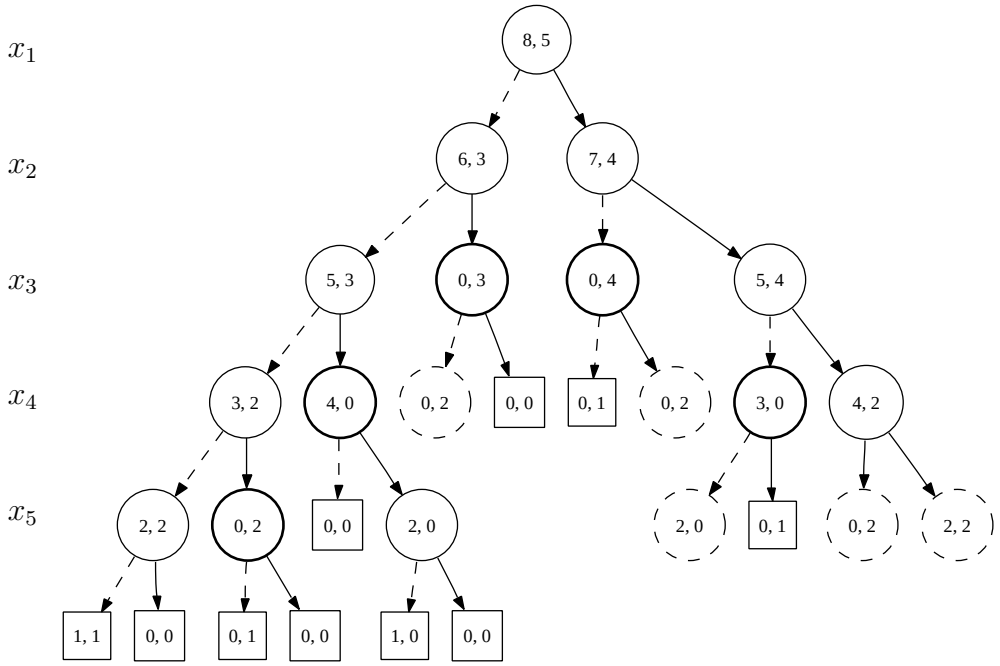

Figure 4.2: Callgraph for `apply` $(\wedge)$ $((x_1 \wedge x_3) \vee (x_2 \wedge x_3))$ $((x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4))$

In a strict language this idea saves the traversal of an ROBDD. In a lazy language it saves the evaluation of this ROBDD. The function `error :: String -> a` causes a runtime error if it is evaluated. The `String` is printed on the screen. The application `apply andM Zero (error "second arg")` yields `Zero`. That is, the second argument is not evaluated. Note that the application `apply andM (error "first arg") Zero` yields an error. Haskell evaluates the arguments of a function from left to right. `andM` checks whether the first argument is a `Zero` leaf, i.e., it has to evaluate the first argument to Head Normal Form. This raises the error.

We assume an abstract data type `ApplyMap` that provides the functions `emptyApplyMap`, `insertApplyMap` and `lookupApplyMap`.

To make as less look-ups as possible we only look up a pair of `NodeId`s if at least one of the nodes is *shared*. That is, one of the nodes is part of an `OBDD` whose root node is a reference. All these nodes are at least twice in the `OBDD`. The original nodes are the leftmost and they have therefore been visited already. If none of them is *shared* the pair of `NodeId`s cannot be memoized. We always deconstruct at least one of the outermost constructors. The `OBDD` contains at most one node with a specific `NodeId` that is no reference and furthermore contains no cycles. Therefore a pair of `NodeId`s cannot be memoized if both nodes are not *shared*.

```
apply :: (Maybe Bool → Maybe Bool → Maybe Bool) → ROBDD
  → ROBDD → ROBDD
apply op robdd1 robdd2 =
  let ROBDD obdd1 _ = robdd1
      ROBDD obdd2 _ = robdd2
  in
  fst (apply1 False emptyApplyMap emptyRevMap obdd1 obdd2)
 where
  apply1 b applymap revmap obdd1 obdd2 =
   case op (obddToBool obdd1) (obddToBool obdd2) of
        Nothing    → apply2 b applymap revmap obdd1 obdd2
        Just False → (ROBDD Zero revmap, applymap)
        Just True  → (ROBDD One revmap, applymap)

  apply2 b applymap revmap obdd1 obdd2
    | b || isRef obdd1 || isRef obdd2 =
      case lookupApplyMap obdd1 obdd2 applymap of
           Nothing   → apply3 True applymap revmap obdd1 obdd2
           Just obdd → (ROBDD obdd revmap, applymap)
    | otherwise = apply3 b applymap revmap obdd1 obdd2

  apply3 b applymap revmap obdd1 obdd2 =
    let (low1, high1, var, low2, high2) = next obdd1 obdd2
        (ROBDD low lowRevmap, lowApplymap) =
                   apply1 b applymap revmap low1 high1
        (ROBDD high highRevmap, highApplymap) =
                   apply1 b lowApplymap lowRevmap low2 high2
        robdd@(ROBDD obdd _) =
                   rOBDD low var high revmap highRevmap
    in
    (robdd,insertApplyMap obdd1 obdd2 (setRef obdd) highApplymap)
```

There are cases in which an application is memoized although only one of the root nodes of the argument `OBDD`s is a *shared* node. Figure 4.4 shows the callgraph of the application of the boolean operator $\wedge$ to the ROBDDs for the expression $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and for the single variable x_4. The node labeled $2, 2$ occurs twice in the callgraph. That is, we have to look up the second application namely the dashed node labeled $2, 2$. This node represents the application of `apply` to two `OBDD`s that are both rooted at the `NodeId` 2. The first `OBDD` is a reference, the second one is none. That is, the first node is *shared* the second one is not *shared*. Thus there are cases in which an application is memoized although only one of the arguments is a *shared* node. If we would only perform a look-up in the *apply map* if the two root node of the `OBDD`s are both *shared* less `NodeId`s would be evaluated. This would be an advantage for the laziness. It is rarely the case that an application is memorized if only one of the arguments is *shared* but measurements showed that it heavily decreases performance if we only look up applications where both nodes are *shared*. This is due to the fact that a look-up

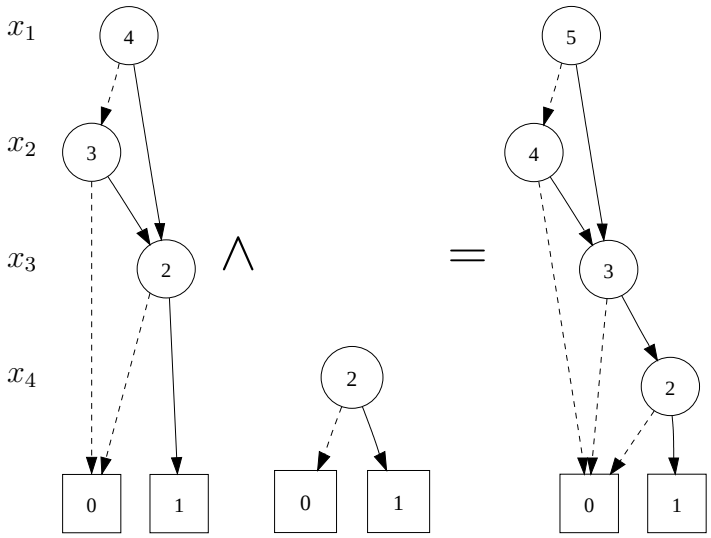

Figure 4.3: ROBDDs for the expressions $(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and x_4 and the result of applying $\wedge$

takes a logarithmic amount of time. The application of `apply` to two `OBDD`s is linear in the number of pairs of `NodeId`s of the two argument `OBDD`s. Thus a missed look-up can be very expensive.

If the `Bool` that states whether a node is a reference is `True` all nodes in the sub-`OBDD` that is rooted at this node are *shared*. Therefore we have to look up all applications in the *apply map* of a node of this `OBDD` and any other node. The additional argument of type `Bool` that is passed to `apply1` takes care of this.

4.1.1 Laziness

The application `apply andM Zero (error "second arg")` yields `Zero`. That is, we do not have to evaluate the second argument if the first is `Zero` and we apply $\wedge$. Independently of the size of the second argument of `apply` the `Zero` leaf is yielded immediately. For example the application `apply andM Zero (integer n)` is in $O(1)$ for every `n`. An implementation in a strict language always has to construct the second argument even though it is not needed. That is, there are applications of `apply` that would not terminate in a reasonable amount of time in a strict implementation while the Haskell implementation yields the result immediately. All applications of `apply` have this property. For example in Figure 4.2 the node labeled $4, 0$ yields `Zero` without evaluating the second argument. This does not guarantee that the second argument is never evaluated. There are other dependencies that can cause its evaluation. For example a look-up in the *apply map* causes the evaluation of the `NodeId`s of the inserts in the *apply map* that are delayed. The evaluation of the `NodeId`s causes the evaluation of parts of the `OBDD`.

The application `apply andM (error "first arg") Zero` raises an error. The boolean

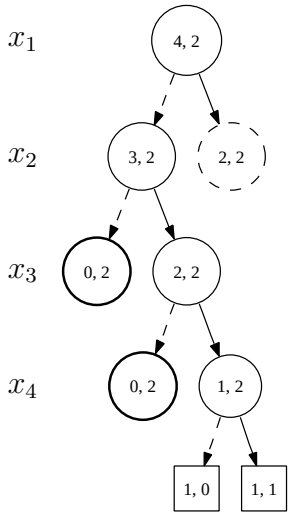

Figure 4.4: Callgraph for `apply` $(\wedge)$ $((x_1 \wedge x_3) \vee (x_2 \wedge x_3))$ x_4

operator `andM` checks whether the first argument is a `Zero` leaf and therefore causes the evaluation of the first argument to Head Normal Form. The application `apply andM (integer n) Zero` does not yield the result in an amount of time that is independent of `n`. The first argument is evaluated to Head Normal Form. Section 3.3.4 already stated that this causes the evaluation of the leftmost path to a node with two leaves and all parts left of it. These are linear many nodes whereas an implementation in a strict language would construct the whole ROBDD, i.e., would evaluate exponentially many nodes.

4.1.2 Measurements

Figure 4.5 shows measurements of the construction of some ROBDDs using `apply`. We apply the functions `anySat` and `eval` to the constructed ROBDDs and measure the construction together with this application. Because of the laziness we cannot separate these two parts. The efficiency of the construction depends on the evaluation that is caused by the consumer function. If the consumer function causes the evaluation of small parts of the ROBDD great parts of the ROBDD are not constructed. All the expressions that are used in this measurement except for `hole8.cnf` are satisfiable.

The function `eval` is a structural equality of an `OBDD` with itself. We use this function to cause the evaluation of the whole result `OBDD`. This illustrates the worst case in respect of the use of laziness. No function causes the evaluation of more nodes of the intermediate and final results than `eval`. We use this function to check whether the implementation with relaxed *no-redundancy* can compete with the implementation with full *no-redundancy* if it cannot benefit from laziness. We could use the `show` function to cause the evaluation but `show` would consume additional heap memory for the `String`

that is yielded. We would not be able to distinguish the memory usage of the `show` application and of the construction.

To check whether the laziness is of any advantage for the construction we compare the application of `anySat` to an relaxed ROBDD with that to an ROBDD with full *no-redundancy*. The application of `anySat` to an satisfiable ROBDD causes the evaluation of just a part of the ROBDD. The size of this part depends on the structure of the ROBDD.

Expression	Operation	No-Redundancy	Time	Memory	Eval. Constr.
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
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
HWB 14	anySat	Relaxed	6.92	1,359,474,756	2044822
		Full	13.82	2,407,651,020	2903684
	eval	Relaxed	13.22	2,232,152,640	2903684
		Full	13.84	2,407,684,808	2903684
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 4.5: Construction of some ROBDDs using `apply`

The implementation with the relaxed *no-redundancy* property is in all measurements better then the implementation with full *no-redundancy* when we apply `anySat`. The values for consumed time, consumed heap memory and the number of evaluated constructors are all less than these values for the implementation with full *no-redundancy*. 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` for satisfiable boolean expressions. The number of evaluated

constructors highly depends on the structure of the boolean expression. For example if we apply $\wedge$ to two ROBDDs where the leftmost variable binding that evaluates both ROBDDs to *true* is far right, i.e., binds many variables to *true* we have to evaluate great parts of the ROBDDs even when we apply `anySat` to the result. The worst case for the laziness is an application to an unsatisfiable ROBDD. This is naturally totally strict, i.e., it has to evaluate the whole ROBDD. This ROBDD is a single leaf but it causes the complete evaluation of all the intermediate results. The expression `hole8.cnf` is unsatisfiable. The application of `eval` and `anySat` evaluate the same number of constructors. This is always the case for unsatisfiable expressions. The application of `eval` to the relaxed ROBDD for the boolean expressions `Queens 8` and `hole8.cnf` cause the evaluation of less constructors in the implementation with full *no-redundancy*. This seems to be odd because we have expected the relaxed ROBDD to have more nodes than the one with full *no-redundancy*. Here the relaxed implementation benefits from the *don't care* idea. Some parts of the intermediate results are not evaluated because they are not needed to determine the result of an `apply` application. The full *no-redundancy* property causes the complete evaluation of all intermediate ROBDDs. Even the sub-ROBDDs that are not evaluated by the relaxed *no-redundancy* are evaluated by the full *no-redundancy* implementation.

When `eval` is applied to the relaxed ROBDD for the expression `uf20-002.cnf` more constructors are evaluated than by the implementation with full *no-redundancy*. For `uf20-002.cnf` the relaxed ROBDD has 59 while the ROBDDD with full *no-redundancy* has 57 nodes. In all the other examples the number of nodes of the result ROBDDs are the same for both implementations.

We cannot explain the differences in the running times of `eval` on an relaxed ROBDD and an ROBDD with full *no-redundancy* for the expression `Integer2 1000`. In all tests for the `Integer2` expression the running time for the `eval` measurement with the relaxed *no-redundancy* is about 35% percent worse than the running time of the implementation with full *no-redundancy*. This is surprising cause the number of evaluated constructors are the same for both implementations.

4.1.3 Complexity

There are $|f||g|$ pairs of `NodeId`s of the ROBDDs f and g. If the sets of variables of the two ROBDDs are disjunct there is one application of `apply` for every pair of `NodeId`s. All the other applications are looked up. If the variables of the two ROBDDs are not disjunct there are less applications. There is one look-up and insert in the *apply map* and an application of `rOBDD` for every application of `apply`. The complexity of the look-up and insert depends on the implementation of the *apply map*. The measurements in Section 5 show that the best implementation for the maps is a Braun Tree or a Patricia Tree. The complexity of insert and look-up of these structures is logarithmic in the size of the key. We use a continuous numbering for the `NodeId`s and can estimate the logarithm of the size of the key by the logarithm of the elements in the map, i.e, the number of pairs. Therefore the worst case running time of `apply` is in $O(log(|f||g|)|f||g|)$. That is, we get an additional logarithmic factor in comparison to an imperative implementation.

4.2 Restrict

The `restrict` operation substitutes one variable in the boolean expression by *true* or *false*. On an ROBDD this operation replaces all nodes with this variable by its low and high successor respectively. We have to guarantee that equal sub-trees are restricted only once. We use a map that memoizes the results of the applications of `restrict`. We assume an abstract data type `MemoMap` that is polymorphic over the type of the entries in the map. We use this data type for all the memoization and the entries in other applications have another type than the one for `restrict`. This ADT provides the functions `emptyMemoMap`, `insertMemoMap` and `lookupMemoMap`.

```
restrict :: Var → Bool → ROBDD → ROBDD
restrict var' b (ROBDD obdd _) =
  fst (restrict1 emptyMemoMap emptyRevMap odd)
 where
  restrict1 memomap revmap Zero = (ROBDD Zero revmap, memomap)

  restrict1 memomap revmap One = (ROBDD One revmap, memomap)

  restrict1 memomap revmap obdd
    | var<var' = restrict2 memomap revmap obdd
    | var==var'   = if b
                      then reduce memomap revmap (getHigh obdd)
                      else reduce memomap revmap (getLow obdd)
    | otherwise = reduce memomap revmap obdd
   where
    var = fromVar (getVar obdd)

  restrict2 memomap revmap obdd
   | isRef obdd =
     case lookupMemoMap (getId obdd) memomap of
          Nothing   → restrict2 memomap revmap (unsetRef obdd)
          Just obdd → (ROBDD obdd revmap, memomap)
   | otherwise =
     let (ROBDD low lowRevmap, lowMemomap) =
                restrict1 memomap revmap (getLow obdd)
         (ROBDD high highRevmap, highMemomap) =
                restrict1 lowMemomap lowRevmap (getHigh obdd)
         robdd@(ROBDD obdd' _) =
                rOBDD low var high revmap highRevmap
     in
     (robdd
     , insertMemoMap (getId2 obdd') (setRef2 obdd') highMemomap)
   where
    var = fromVar (getVar obdd)
```

The function `getVar` yields the variable of a node wrapped in a constructor named `Var`. If `getVar` is applied to a leaf it yields a nullary constructor. We do not use the `Maybe` type instead because we want to make the type an instance of the type class `Ord` and want to use another implementation than it is used for the `Maybe` type. The function `fromVar` wipes the additional constructor off like `fromJust`.

The restriction of a leaf yields the same kind of leaf. If we restrict a node we have to compare its variable to the variable we are restricting. If the variable is still smaller we check whether the node is a reference. In this case we look it up in the *memo map*. If it is not a reference we continue descending the ROBDD. We have to reduce the result because `restrict` can produce nodes that can be shared after the restriction and were not before. Therefore we reconstruct the ROBDD by using `rOBDD`. If the look-up of a reference fails we have chucked away the original node. In this case we change the reference into an original node by applying `unsetRef`. This idea was introduced in Section 3.2.

If the variable of the node is greater than the one we are restricting we do not have to look for the variable any longer. We apply the function `reduce` to the `OBDD`. This function reconstructs the ROBDD and adds all nodes to the *reverse map* and all partial results to the *memo map*. The only difference between `reduce` and `restrict` is that `reduce` does not check the variable numbers.

If the correct variable is found the low and high successor is yielded respectively. We have to add all nodes of this `OBDD` to the *reverse map* and the *memo map* via `reduce`.

4.2.1 Laziness

The function `restrict` takes an ROBDD and yields a new ROBDD. This section investigates which parts of the argument are evaluated because of `restrict`. That is, we check which parts of the argument are evaluated when only parts of the result of `restrict` are evaluated.

We check which parts of the argument `OBDD` are evaluated when `restrict` is applied to it and `anySat` to the result of `restrict`. We can compare these parts with the parts that are evaluated when `anySat` is applied without an application of `restrict`. This gives use a hind which parts of the `OBDD` are evaluated because of the additional `restrict` application. We distinguish two cases of restrictions.

Let $(x_i, b_i) \dots (x_n, b_n)$ be the path from the root node to the leftmost node with two leaves. Let x_k be the variable that is restricted and b be the boolean value x_k is restricted to. We distinguish the cases $b = b_k$ and $b \neq b_k$. That is, we preserve the path to the leftmost node with two leaves in the first case and destroy it in the second case.

We restrict variable x_3 to `True` in the ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. The left ROBDD in Figure 4.6 shows this ROBDD. The center ROBDD shows the result of the restriction. The bold parts of the ROBDDs are the paths from the root node to the leftmost node with two leaves. The restriction preserves this path and just shortens it by one variable. That is, the path to the leftmost node with two leaves in the result of `restrict` is the same as in the argument.

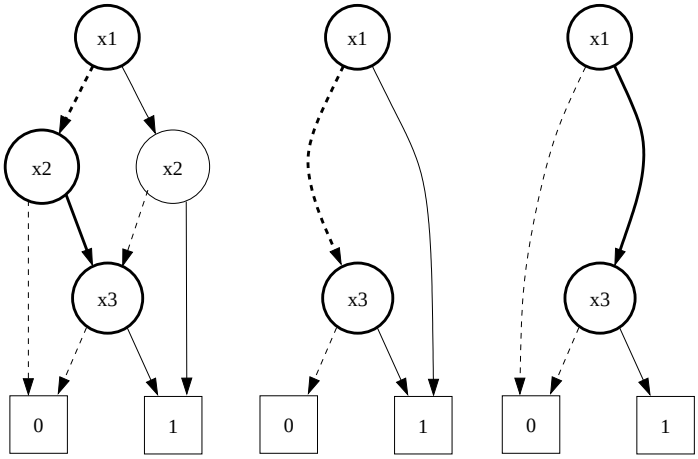

Figure 4.6: An ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ (left), the result of `restrict` x_2 `True` (center) and the result of `restrict` x_2 `False` (right)

We apply `anySat` to the result of the restriction and observe the argument `OBDD`. The left part of Figure 4.7 shows this observation. The right part shows the observation of the argument `OBDD` when applying `anySat` without an application of `restrict`. The two observations prove the considerations. An application of `anySat` to the result of `restrict` causes the evaluation of the same parts as an application of `anySat` without a `restrict` application.

It is very difficult to predict the evaluation of the argument of `restrict` that is caused by an application of `anySat` to the result if $b \neq b_k$ holds. In this case the path to the leftmost node with two leaves is destroyed. There are cases in which a new node with two leaves is constructed by the restriction. Figure 4.8 shows a part of an ROBDD. If we restrict variable x_m to `True` in this ROBDD a new leftmost node with two leaves is constructed. The right part of the figure shows the result of the restriction. The new leftmost node with two leaves is on the path to the old one. Therefore an application of `anySat` to the result of `restrict` causes the evaluation of the same parts of the argument as without the `restrict` application.

We take a look at another example where $b \neq b_k$ holds but no new leftmost node with two leaves is constructed. In the ROBDD for the expression $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ variable x_2 is restricted to `False`. The right ROBDD in Figure 4.6 shows the result of this restriction. Figure 4.9 shows the observation of this restriction. We throw away the high successor of the left node labeled x_2. Therefore the path to the leftmost node with two leaves is no more existent in the result of `restrict`. That is, the application of `anySat` causes the evaluation of nodes that are not evaluated by an application without `restrict`. An application of `anySat` to the result of the restriction causes the evaluation of the bold parts of the right ROBDD in Figure 4.6 and all parts left of it. This causes the evaluation of the high successor of the root node in the argument of `restrict`. The observation proves this statement. The high successor of the root node is evaluated. This part is not evaluated in the right observation in Figure 4.7 that observes an application

```
Just [(1,False),(3,True)]

>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    -
                    False)
              -
              False)
        1
        -
        -
        False)
```

```
Just [(1,False),(2,True),(3,True)]

>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    -
                    False)
              -
              False)
        1
        -
        -
        False)
```

Figure 4.7: Observation of the `OBDD` structure when applying `anySat . restrict` x_2 `True` (left) and `anySat` (right)

of `anySat` without an application of `restrict`. The high successor of the root node is a reference. The evaluation of this node causes the evaluation of the original node that is positioned in the part that is chucked away by the restriction.

4.2.2 Measurements

We want to measure the effect of a restriction to the laziness. We measure the application of `anySat` to six restrictions of three ROBDDs each. The effect of a `restrict` application on the laziness depends highly on the variable that is restricted. We present measurements of restrictions of the variables x_1, $x_{\frac{n}{2}}$ and x_n where n is the number of variables in the ROBDD. That is, we restrict the top variable the bottom variable and the one in the middle. Each variable is restricted to `True` and `False`.

If we apply `eval` to the restricted ROBDD the results do not differ highly. The whole ROBDD is evaluated in this case and the complexity of `restrict` is independent from the variable that is restricted. An application of `anySat` to an ROBDD with full *no-redundancy* shows the same behavior. That is, there are nearly no differences in the restrictions of the different variables. The measurement of the implementation with relaxed *no-redundancy* always performs better or at least as good as the implementation with full *no-redundancy* in respect to time and memory consumption. Therefore we only present measurements for the relaxed *no-redundancy* implementation.

In addition to the restrictions we measure the application of `anySat` without an application of `restrict`. This is a lower bound for the results of the application of `anySat`

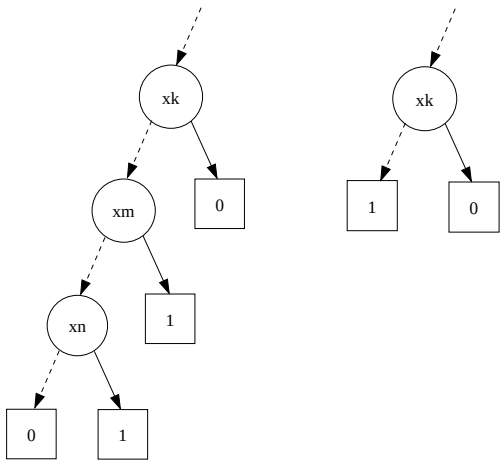

Figure 4.8: A part of an ROBDD (left) and the result of `restrict` x_m `False` (right)

with a restriction. That is, an application of `anySat` with an additional restriction can at most be as lazy as this application. Besides this we measure an application of `eval` to a restriction of a variable that is not part of the ROBDD. These measurements are labeled `100 False (eval)`, i.e., we restrict variable x_{100} to `False` and apply `eval` to the result. The function `eval` causes the evaluation of the whole ROBDD. The additional restriction is important because `restrict` reconstructs the ROBDD. Therefore a restriction always increases the number of evaluated constructors and the memory consumption. The application of `eval` in combination with a restriction is an upper bound for the performance of the applications of `anySat` because it causes the evaluation of all nodes. The number of evaluated constructors and therefore the time and memory consumption of all the applications of `restrict` will be somewhere between the values of the lower and the upper bound. These bounds make it easier to rate the measurements that are presented here.

The laziness of a restriction depends on the fact whether the path to the leftmost node with two leaves is destroyed or not. Additionally the laziness depends on the variable that is restricted. The smaller the variable, i.e., the higher the node the more nodes are evaluated additionally.

The restriction of the bottom variable always destroys the path to the leftmost node with two leaves because if chucks away one of the two leaves. In the measurements of `Queens 8` the restrictions of all other variables to `False` preserve the path. In the measurements of `Integer 16` the restrictions of variable x_1 to `False` and x_{16} to `True` preserve the path. The restrictions of variable x_1 to `False` and x_7 to `True` in the expression `HWB 14` preserve the path. The results for the restrictions that preserve the path are very close to the results of an application of `anySat` without a restriction. These measurements differ slightly because the sub-`OBDD` that is chucked away is not evaluated and the size of this `OBDD` is different for different variables.

The restrictions that destroy the path cause the evaluation of more constructors than the restrictions that preserve the path. The smaller the variable the greater the part

```
Just [(1,True),(3,True)]

>>>>>>> Observations <<<<<<

testExp4
  (OBDD (OBDD Zero
              2
              (OBDD Zero
                    3
                    One
                    2
                    False)
              _
              False)
        1
        (OBDD (OBDD Zero
                    3
                    One
                    2
                    True)
              2
              One
              _
              False)
        _
        False)
```

Figure 4.9: Observation of the restriction of variable x_2 to `False`

that is additionally evaluated.

The main observation of this section is that an application of `restrict` does not categorically destroy the laziness. That is, `restrict` can be used without the fear that the laziness of the construction is lost.

4.3 Equality Check

This chapter discusses different implementations of the equality check. The main problem here is that ROBDDs with relaxed *no-redundancy* are no more canonical. The equality check of ROBDDs makes use of the canonical representation. Therefore the definition of an equality check for the ROBDD implementation with relaxed *no-redundancy* property requires some work.

Expression	Restriction	Time	Memory	Eval. Constr.
Queens 8	100 False (eval)	32.22	3,674,966,992	2203306
	none	25.28	2,918,337,044	1874446
	1 False	27.02	2,918,358,944	1874509
	1 True	34.06	3,629,573,656	2200563
	32 False	27.08	2,918,360,172	1874509
	32 True	28.08	3,035,483,776	1924515
	64 False	25.50	2,953,945,216	1874509
	64 True	27.54	3,172,832,228	1969764
Integer 16	100 False (eval)	6.72	770,216,508	491526
	none	0.00	202,064	214
	1 False	0.00	207,180	230
	1 True	3.68	390,389,092	221227
	16 False	0.00	208,880	231
	16 True	0.00	207,700	230
	32 False	0.00	275,200	288
	32 True	0.00	207,800	230
HWB 14	100 False (eval)	13.24	2,232,768,912	2904328
	none	6.92	1,359,474,756	2044822
	1 False	7.24	1,353,462,308	2044835
	1 True	12.84	2,231,921,548	2903368
	7 False	7.70	1,422,135,936	2122126
	7 True	6.94	1,359,479,276	2044835
	14 False	6.86	1,359,543,804	2044909
	14 True	6.92	1,359,479,536	2044835

Figure 4.10: Measurements of the restriction of several variables in several ROBDDs

4.3.1 Full No-Redundancy Property

First we introduce the equality check for ROBDDs with full *no-redundancy* property. For this implementation the equality check is a check for isomorphy.

```
instance Eq OBDD where
  (OBDD _ _ _ nodeId1 True) == (OBDD _ _ _ nodeId2 True) =
    nodeId1==nodeId2

  (OBDD l1 var1 h1 _ False) == (OBDD l2 var2 h2 _ False) =
    var1==var2 && l1==l2 && h1==h2

  Zero == Zero = True

  One == One = True

  _ == _ = False
```

To compare two references we compare their `NodeId`s. These are equal if the two ROBDDs are equal because they are constructed using a deterministic algorithm. If we compare two `OBDD`s that are no references we compare their variable numbers and their low and high successors. Two `Zero` and two `One` leaves are equal. All other `OBDD`s are not equal. This equality check is in $O(min\{|f|, |g|\})$. We stop the traversal of the ROBDDs at the latest when we have traversed the smaller ROBDD. We cannot use this equality check for relaxed ROBDDs. If we add redundant nodes to an ROBDD we get an ROBDD that represents the same boolean function but is not isomorphic to the one without these nodes. That is, there are relaxed ROBDDs that are equal but not structural equal.

4.3.2 Relaxed No-Redundancy Property

One way to check the equality of two relaxed ROBDDs is to reduce the ROBDDs with the full *no-redundancy* property and apply the isomorphy check to the results.

The term $r(n)$ where n is the number of variables denotes the worst case number of redundant nodes that an ROBDD with relaxed *no-redundancy* additional contains per node in the ROBDD with full *no-redundancy*. Let f and g be two ROBDDs with full *no-redundancy* property. Let f' and g' be ROBDDs with relaxed *no-redundancy* property that represent the same boolean functions. These ROBDDs have $r(m)|f|$ and $r(n)|g|$ nodes in the worst case where m and n are the numbers of variables of f and g respectively. We can reduce these ROBDDs in $O(log(|f'||f'|))$ and $O(log(|g'|)|g'|)$. The equality check of the reduced ROBDDs is in $O(min\{|f|, |g|\})$. We can estimate $log(|f'|)|f'| + log(|g'|)|g'|$ and $min\{|f|, |g|\}$ by $max\{log(|f'|)|f'|), log(|g'|)|g'|\}$. That is, this equality check for relaxed ROBDDs is in $O(max\{log(|f'|)|f'|, log(|g'|)|g'|\})$.

This implementation of the equality check is strict, i.e., it causes the complete evaluation of both ROBDDs even if they are unequal. The result of the reductions are ROBDDs with full *no-redundancy* property. If we evaluate these to Head Normal Form we have to evaluate the whole ROBDDs.

Another way of checking equality is using `apply` with the boolean operator $\Leftrightarrow$ and check whether the result is the boolean constant *true*, i.e, the `One` leaf. The worst case running time of this operation is in $O(log(|f'||g'|)|f'||g'|)$. To check whether the result of `apply` $\Leftrightarrow$ is a `One` leaf it must be evaluated to Head Normal Form. This causes the evaluation of all nodes on the leftmost path to a node with two leaves and left of it. It is very difficult to argue which parts of the argument ROBDDs are evaluated. This depends highly on the structure of the arguments.

The worst case for the laziness, i.e., the most nodes are evaluated, if the two ROBDDs are equal or if one of them is the negation of the other. In these cases the result of `apply` $\Leftrightarrow$ is the `Zero` or the `One` leaf and we have to evaluate both arguments completely.

4.3.3 Measurements

Figure 4.11 shows some measurements of the equality check of equal ROBDDs using the three different kinds of equality checks. Figure 4.12 shows some measurements of the equality check of unequal ROBDDs. We only expose the more significant examples

here. The first two columns report the expressions that are compared. The third one reports the type of equality check that is used. $Eq1$ is the equality check using `reduce` of relaxed ROBDDs. $Eq2$ is the equality check using `apply` of relaxed ROBDDs. $Eq3$ is the check for isomorphy of ROBDDs with full *no-redundancy* property. The third and fourth column show the time and memory usage of the applications respectively. The fifth column reports the number of constructors that are evaluated in total, i.e., in both ROBDDs together.

Fst Argument	Snd Argument	Check	Time	Memory	Eval. Constr.
Queens 6	Queens 6	Eq1	3.44	429,069,396	308832
		Eq2	3.48	428,944,256	308568
		Eq3	3.46	434,143,972	311444
Integer 16	Integer 16	Eq1	12.78	1,521,499,032	983052
		Eq2	10.34	1,196,614,376	655378
		Eq3	8.94	1,040,711,296	655378
Integer2 500	Integer2 500	Eq1	8.56	897,237,676	757494
		Eq2	8.78	895,615,668	754496
		Eq3	6.44	900,722,276	754496
uf20-02.cnf	uf20-02.cnf	Eq1	1.08	143,467,748	101982
		Eq2	1.08	143,342,756	101860
		Eq3	1.04	140,568,616	99036

Figure 4.11: Equality check of equal ROBDDs

$Eq1$ basically causes the evaluation of more constructors because `reduce` reconstructs the two argument ROBDDs. The reconstruction is most significant in the measurement of `Integer 16` because the resulting ROBDD is very big in this example. In the measurement of `Queens 6` the $Eq1$ check evaluates even less constructors than $Eq3$ does. The number of additional constructors that are used by the reduction is very small in this example. As we have already observed in Section 4.1.2 the construction of a `Queens` expression highly benefits from the *don't care* idea. Many constructors of the intermediate results are not evaluated because of the *don't cares*. Because $Eq3$ is applied to an ROBDD with full *no-redundancy* there are no constructors that are not evaluated because of the *don't cares*. The equality checks $Eq1$ and $Eq2$ do not cause the evaluation of these constructors because they are applied to ROBDDs with relaxed *no-redundancy*.

In the measurements of the expression `uf20-02.cnf` $Eq3$ causes the evaluation of less constructors than the other two equality checks. The relaxed ROBDD simply contains more nodes. We have already observed this in Section 4.1.2. This causes the evaluation of more constructors when we compare two ROBDDs with relaxed *no-redundancy* than two ROBDDs with full *no-redundancy*.

The running times for $Eq2$ are better than we expected. The worst case running times stated that $Eq2$ is quadratic in the number of nodes with an additional logarithmic factor while $Eq1$ is linear in the number of nodes. As stated in Chapter 4.1 the running time

of `apply` is only quadratic if the sets of variables of the two ROBDDs are disjunct. In the comparison of two equal ROBDDs the sets of variables are equal. Thus, the running time of `apply` is linear in the size of the ROBDD because `apply` traverses the two ROBDDs parallel. Note that this is only true for structural equal ROBDDs. Because of the relaxing of the *no-redundancy* there are ROBDDs that are equal but not structural equal.

Let $f1$ and $f2$ be two ROBDDs with relaxed *no-redundancy* that are equal but not structural equal. Let f be the ROBDD with full *no-redundancy* that represents the same boolean function as $f1$ and $f2$. The term $r(m)$ where m is the number of variables in f describes the worst case number of redundant nodes per node. That is, it is $|f1| \leq r(m)|f|$ and $|f2| \leq r(n)|f|$ where m and n are the number of variables in $f1$ and $f2$ respectively.

We use the equality check $Eq2$ to compare the ROBDDs $f1$ and $f2$. Let x be any of the redundant nodes that can be in an ROBDD with relaxed no-redundancy. In the equality check of $f1$ and $f2$ there are three cases. In the following we have to keep in mind that the equality check $Eq2$ on f with itself is a parallel descending of the ROBDDs. That is, all application of `apply` in this equality check are applications to equal nodes. If neither $f1$ nor $f2$ contains `x` this node has no influence on the running time of the equality check.

If both ROBDDs contain `x` there is an application to these two redundant nodes. Because the variable numbers of the nodes are equal they are both descended. There are two applications of `apply` to the two successors of the redundant node. Because the successors are equal, i.e., they have the same `NodeId` the second application is looked up in the *memo map*. That is, `apply` performs one additional descent and one look up.

If only one of the ROBDDs contains `x` there is an application to this node and another node `y`. The variable number of the redundant node is less than the number of `y`. Therefore there are two applications of `apply` each to the successor of the redundant node and `y`. The successors of the redundant node are equal and the second application is therefore looked up in the *memo map*.

Every redundant node causes at most one additional descend of a node and one look-up in the *apply map*. Therefore the worst case running time of an equality check using $Eq2$ of two equal ROBDDs with relaxed *no-redundancy* is in $O(log(r(m)|f|)r(m)|f|)$. That is, it is linear in the worst case size of the ROBDD with relaxed *no-redundancy* property with an additional logarithmic factor.

In all examples except for `uf20-02.cnf` the relaxed ROBDD contains no redundant nodes, i.e., $|f1| = |f2| = |f|$. Therefore $Eq2$ is in $O(log(|f|)|f|)$ in these cases. The equality check $Eq3$ is in $O(|f|)$ for all examples. The running times of $Eq2$ are worse than the ones of $Eq2$ in the examples with no redundant nodes because of the additional logarithmic factor in the running time of $Eq2$.

If we compare unequal ROBDDs the implementation using `apply` is far better then the isomorphy check. This is due to the laziness of this equality check. We only have to evaluate the result of `apply` $\Leftrightarrow$ to Head Normal Form. All examples except for the last one cause the evaluation of about one third of the constructors that are evaluated by $Eq3$.

Fst Argument	Snd Argument	Check	Time	Memory	Eval. Constr.
Queens 6	Integer2 1000	Eq1	20.04	2,074,778,364	1661913
		Eq2	2.58	472,308,836	638944
		Eq3	14.80	2,086,692,328	1660220
Integer2 500	Integer2 1000	Eq1	22.86	2,308,865,908	1886244
		Eq2	1.72	368,051,084	634247
		Eq3	16.52	2,319,984,904	1881746
Integer 16	Integer2 1000	Eq1	24.94	2,617,326,780	1999023
		Eq2	1.18	294,056,264	506721
		Eq3	17.56	2,386,309,608	1832187
uf20/uf20-02.cnf	Integer2 1000	Eq1	14.84	1,932,015,160	1557489
		Eq2	1.30	299,138,080	511191
		Eq3	13.90	1,939,907,300	1554016
Integer2 1000	uf20/uf20-02.cnf	Eq1	14.82	1,932,015,160	1557489
		Eq2	14.30	1,863,295,600	1509176
		Eq3	13.70	1,939,907,264	1554016

Figure 4.12: Equality check of unequal ROBDDs

The last two measurements in Figure 4.12 show that the efficiency of the lazy equality check depends on the order of the arguments. The last two measurements are the same equality checks with swaped arguments. While the two other methods yield the same time and memory consumption and number of evaluated constructors the results for $Eq2$ differ highly. One of the applications needs 14.30 seconds while the other one needs only 1.30 seconds for the check. This is due to the fact that Haskell evaluates function arguments from left to right. For example the function `rOBDD` has to check whether one of its arguments is a leaf. Haskell first evaluates the leftmost argument to Head Normal Form. If this is not leaf it does not check the other argument. Therefore it can depend on the order of the arguments which parts of a data structure are evaluated. It would be interesting to investigate the lazy running time of the equality check that uses `apply`. This is supposed to be very complex because the analysis of the evaluation what parts of the two ROBDDs are evaluated is far from trivial. If the two ROBDDs are equal the running time is linear in the number of nodes of the ROBDD with an additional logarithmic factor. In the measurements of the check of unequal ROBDDs the equality check is faster than the one that is linear in the number of nodes. That is, it would be interesting whether an overall lazy running time of this equality check would be linear with an additional logarithmic factor.

4.4 Consumer Functions

All the consumer functions, i.e., functions that take an ROBDD and yield a value and not an ROBDD have the same general structure. Examples for this kind of functions are

`anySat`, `allSat`, `satCount` and `evaluate`. All these operations work like a fold with the additional condition that partial results are memoized and looked up when a reference is visited. We implement a `fold` function that uses a *memo map* that maps the `NodeId`s of all processed nodes to the result of the processing. We assume an abstract data type called `MemoMap` that is polymorphic over the type of the entries in the map like we used it for the implementation of `restrict`. This ADT provides the functions `emptyMemoMap`, `insertMemoMap` and `lookupMemoMap`.

```
fold :: (a → Int → 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 obdd
    | isRef obdd =
      case lookupMemoMap obdd memomap of
           Nothing → error ("fold: no original node for the"
                            ++ " NodeId " ++ show (getId obdd))
           Just v  → (v, memomap)
    | otherwise  =
      let (lowV, lowMemomap) = fold' memomap (getLow obdd)
          (highV, highMemomap) = fold' lowMemomap (getHigh obdd)
          v' = f lowV (fromVar (getVar obdd)) highV
      in
      (v', insertMemoMap obdd v' highMemomap)
```

With this abstraction it is easy to define a couple of functions. We just have to define two basic cases and the function that is used to combine two partial results. We do not have to worry about the memoization. The `fold` function takes care that the result for equal sub-`OBDD`s are computed only once.

```
anySat :: ROBDD → Maybe Binding
anySat (ROBDD obdd _) = fold sat Nothing (Just []) obdd
 where
  sat Nothing _ Nothing       = error "anySat: redundant nodes"
                                    ++ " on an edge to a leaf"
  sat Nothing var (Just path) = Just ((var,True):path)
  sat (Just path) var _       = Just ((var,False):path)


allSat :: OBDD → [Binding]
allSat = fold sat [] [[]]
 where
  sat low var high =
    map ((var,False):) low ++ map ((var,True):) high
```

```
evaluate :: Binding → ROBDD → Bool
evaluate bs (ROBDD obdd _) =
  fold eval False True obdd
 where
  eval low var high =
    case lookup var bs of
         Just b → if b
                     then high
                     else low
         _      → error ("evaluate:␣binding␣does␣not␣contain"
                    ++ "␣variable␣" ++ show var)


satCount :: ROBDD → Int
satCount (ROBDD obdd revmap) =
  let (var,count) = fold count (0,getNextId revmap+1)
                               (1,getNextId revmap+1) obdd
  in
  2^(var-1) * count
 where
  count (varl,low) var (varh,high) =
    (2^(varl-var-1) * low + 2^(varh-var-1) * high, var)
```

The implementations of `anySat`, `allSat` and `evaluate` are straight forward. Note that `anySat` makes no use of the memoization. It does not evaluate any reference node and therefore never performs a look-up. There is a little overhead in comparison with an implementation that does not use `fold` because of the suspended inserts in the *memo map*. The function `satCount` implements the formula that is presented in Section 1.1. Therefore we need the greatest variable number in an ROBDD. The *reverse map* is enriched with this information. In the smart constructor `rOBDD` we adjust this information every time a node is constructed.

4.5 User Interface

The user interface is inspired by a paper by Nancy A. Day, John Launchbury and Jeff Lewis [11]. They use the type classes `Boolean` and `Var` to define constructor functions for boolean expressions. The type class `Boolean` contains functions that are used to construct a boolean expression without variables. The class `Var` extends this class to the construction of boolean expressions with variables. We use similar type classes and call them `Boolean` and `BooleanExp`.

The types that are members of the class `Boolean` represent boolean expressions without variables. The class provides the constants `bFalse` and `bTrue` and the functions `bNot` and `(∧)`. The class additionally provides the functions `(∨)`, `(⇒)` and `(⇔)` that have default

implementations. The class contains two additional functions namely `bAll` and `bAny`. These functions are the generalizations of `(∨)` and `(∧)` to lists of boolean expressions. These functions are very useful in the construction of boolean expressions that have a regular form like boolean expressions in CNF or DNF. We provide an instance of the type `Bool` to demonstrate the use of this class.

```
class Boolean b where
  bFalse :: b
  bTrue :: b
  bNot :: b → b
  (∧) :: b → b → b
  (∨) :: b → b → b
  x ∨ y = bNot (bNot x ∧ bNot y)
  (⇒) :: b → b → b
  x ⇒ y = bNot x ∨ y
  (⇔) :: b → b → b
  x ⇔ y = (x ⇒ y) ∧ (y ⇒ x)
  bAll :: [b] → b
  bAll [] = bTrue
  bAll bexps = foldr1 (∧) bexps
  bAny :: [b] → b
  bAny [] = bFalse
  bAny bexps = foldr1 (∨) bexps


instance Boolean Bool where
  bFalse = False
  bTrue = True
  bNot = not
  (∧) = (&&)
  (∨) = (||)
```

The members of the type class `BooleanExp` are boolean expressions with variables. This type class just provides one function called `bVar` that takes a variable number and yields a boolean variable with this number. All members of the class `BooleanExp` have to be members of the class `Boolean`.

```
class Boolean b => BooleanExp b where
  bVar :: Int → b
```

The instances of the classes `Boolean` and `BooleanExp` for the type `BExp` that implements boolean expressions are canonical. By defining the functions `(⇒)` and `(⇔)` we replace the default implementations of these functions.

```
instance Boolean BExp where
  bFalse = BFalse
  bTrue = BTrue
  bNot = BNot
  (∧) = BAnd
  (∨) = BOr
  (⇒) = BImp
  (⇔) = BEqu

instance BooleanExp BExp where
  bVar = BVar
```

The instances for the type ROBDD are straight forward. The functions `(⇒)` and `(⇔)` provide default implementations but we reimplement these function because of the efficiency. For details take a look at Section 4.1.

```
instance Boolean ROBDD where
  bFalse = ROBDD Zero emptyRevMap
  bTrue  = ROBDD One emptyRevMap
  bNot = negate
  (∧) = apply andM
  (∨) = apply orM
  (⇒) = apply impM
  (⇔) = apply eqM

instance BooleanExp ROBDD where
  bVar i = ROBDD (OBDD (Zero False) i (One False) 2 False)
                 emptyRevMap
```

Besides the construction of an ROBDD we want to provide additional operations for their manipulation. We extend the class `BooleanExp` to `BooleanPackage`. This type class provides all the functions that are part of an BDD Package. The equality check is not part of this class. We use the type class `Eq` that is part of the Haskell Standard [20] for its implementation. This class provides the single function `(==) :: a → a → Bool`.

An instance of the type class `BooleanPackage` requires the definition of `allSat`. On basis of this function we provide a default implementation for the function `anySat`. In a strict language this would increase the complexity of `anySat`. In a lazy language this implementation is as efficient as an independent implementation of `anySat`. This implementation causes just a little overhead because of the suspended evaluations. Apart from these two functions this type class provides the functions `evaluate`, `restrict` and `satCount`.

```
class (BooleanExp b, Eq b) => BooleanPackage b where
  allSat :: b → [Binding]
  anySat :: b → Maybe Binding
  anySat b =
    case allSat b of
         [] → Nothing
         (x:_) → Just x
  evaluate :: Binding → b → Bool
  restrict :: Var → Bool → b → b
  satCount :: b → Int
```

It is very easy to extend this design. If we want to extend the BDD Package by further operations we just add an interface of these operations to the type class `BooleanPackage`. Another advantage of this user interface is that there is no way to construct an ROBDD that does not fulfill the *sharing* and the relaxed *no-redundancy* property. We do not export the constructors of `ROBDD` and `OBDD`. The only way to construct an ROBDD is using the functions that are provided by the type class `BooleanExp`.

Additionally we can change the concrete implementation of the boolean expressions by changing type annotations. For example if we apply the function show to a member of `BooleanExp` and annotate the type `ROBDD` the expression is displayed as an acyclic graph. We represent the graph just like the `OBDD` data structure, i.e., by a tree with reference edges. If we annotate the type `BExp` to the expression the boolean expression is displayed. We can extend this idea by adding more instances to the type classes to alter the concrete implementation of consumer functions like show or the construction itself.

There is a little drawback of this implementation. We always have to add type annotations when we use boolean expressions. The consumer functions are overloaded. The type checker cannot infer which implementation should be used when we generate a boolean expression and apply a consumer function to it. The only way to solve this would be not to use the type class `BooleanPackage`. If we use different names for the implementations of a consumer function for different representations the type checker knows which construction to use by the type of the consumer function.

5 Implementation of the Maps

This chapter discusses the implementation of the maps that are used in this work. We have to implement three abstract data types, namely `RevMap`, `ApplyMap` and `MemoMap`. Imperative implementations use hash tables for the *reverse* and the *apply map* because they can be very sparse. We do not use hashing here and focus on the map implementation itself. The `MemoMap` is implemented by an array in the imperative implementation.

The *memo map* uses `NodeId`s as keys. The *apply map* uses `(NodeId,NodeId)` and the *reverse map* `(NodeId,Var,NodeId)`. The types `NodeId` and `Var` are type synonyms for integer types.

A look-up in a map that uses a numerical type as keys causes the evaluation of the keys of all delayed inserts. If a look-up in a map is performed the keys of the delayed inserts are evaluated to check whether the insert replaces the entry that is requested. If we use an algebraic data type for the keys it depends on the equality check whether the whole key is evaluated. It depends on the value of the key and the concrete map implementation whether the insert is completely evaluated. That is, if the entry that is looked up is positioned in another part of the map than the entries of the delayed inserts the inserts are only as far evaluated as the look-up takes the same path as the inserts. The inserts are delayed where the paths diverge.

Therefore the laziness of a map implementation has no effect on the number of evaluated constructors in the ROBDD. The only way to cause the evaluation of a part of the ROBDD by the evaluation of a part of the map is the evaluation of a `NodeId`. The entries in the map are never evaluated because of a look-up or an insert. All map implementations cause the evaluation of all keys, i.e., the `NodeId`s, of all delayed inserts. Therefore the same parts of the ROBDD are evaluated because of a look-up no matter which implementation is used. The laziness of the maps that is observed in this chapter only saves the evaluation of parts of the map structure itself.

5.1 Map

The GHC provides an implementation of a finite map. The version 6.4 supports a new implementation called `Map`. This implementation is based on size balanced binary trees and trees of bounded balance [1, 24].

```
data Map k a  = Tip
              | Bin !Size !k a !(Map k a) !(Map k a)

type Size     = Int
```

The exclamation marks in the definition of the `Map` data type are strictness annotations. A strictness annotation states that every application of the corresponding constructor is applied with the strict apply function `$!` instead of the non strict apply function `$`. This strict apply is defined by means of the `seq` function.

```
seq :: a → b → b
seq _|_ b = _|_
seq a   b = b, if a /= _|_
```

This is not a valid Haskell syntax but it points out the semantics of the function. First `seq` checks whether the first argument is bottom. To check this the first argument is evaluated to Head Normal Form. That is, `seq` evaluates its first argument to Head Normal Form and yields its second argument.

The strict apply function `$!` is defined by means of `seq`. It evaluates its second argument to Head Normal Form before it applies the provided function to it.

```
($), ($!) :: (a → b) → a → b
f $  x =          f x
f $! x = x `seq` f x
```

In the case of the `Map` data type the evaluation of a `Map` to Head Normal Form causes the whole `Map` structure to be evaluated because of the strictness annotations. The entries of the `Map` do not have a strictness annotation, i.e., they are not evaluated until they are needed.

We add the numbers `2`, `1` and `3` in this order to an empty map and look up `2`. We observe the data structure that is passed to the look-up.

```
>>> Observations <<<

Map
  (Bin 3 2 2 (Bin 1 1 _ Tip Tip) (Bin 1 3 _ Tip Tip))
```

The observations backup the considerations stated before. The whole structure except for the entries is evaluated.

5.2 FiniteMap

We take a look at the deprecated library `FiniteMap` since this one contains no strictness annotations.

```
data FiniteMap key elt = EmptyFM
                       | Branch key elt Int (FiniteMap key elt)
                                            (FiniteMap key elt)
```

We again add the numbers `2`, `1` and `3` in this order to an empty map and look up `2`. We observe the data structure that is passed to the look-up.

```
>>> Observations <<<

FM
  (Branch 2 2 _ _ _)
```

Only the root node of the `FiniteMap` is evaluated because this one directly contains the key we are looking for. This picture changes if we change the order in which the numbers are inserted from `2,1,3` to `1,2,3`.

```
>>> Observations <<<

FM
  (Branch 2 2 _ (Branch 1 _ 1 EmptyFM EmptyFM) _)
```

Although we look up the `2` which is the key of the root node the node with the key `1` is evaluated. This evaluation is caused by the rotation that preserves the search tree property. The first example inserts the values in a way that no rotation is needed. First we insert the `2`. The `1` is inserted as left successor of `2`. The `3` is inserted to the right of `2`. If we insert the numbers in the order `1,2,3` an insert without a rotation would result in an unbalanced tree. Therefore a rotation is needed which has to evaluate parts of the data structure. The relation between the order of inserts and the evaluation is not trivial. It requires a close understanding of the specific finite map implementation.

5.3 BraunTree

Braun Trees [7] are a long but not widely known data structure. A Braun Tree is a binary tree that is used to save data with positive numerical values as keys. The root node contains the value for the key zero. To look up a key its checked whether the key is even or odd and the left and right successor is taken respectively. On every step down the tree the key is divided by two. If the key reaches zero the element is inserted and looked up respectively. Braun Trees are used to implement purely functional arrays.

```
data BraunTree a = Node a (BraunTree a) (BraunTree a)
```

The empty `BraunTree` in this implementation is an infinite data structure whose entries all `Nothing`. Because of laziness we do not have to bother with extending the tree from time to time. We just generate an infinite data structure and let the lazy evaluation do all the bookkeeping.

```
>>> Observations <<<

Braun
  (Node _ _ (Node _ (Just 2) _))
```

The look-up in the `BraunTree` only evaluates the path to the key we are looking for. On the other side the `BraunTree` is logarithmic in the key while balanced search trees are logarithmic in the number of elements in the tree. That is, the Braun Tree is only as good as a balanced search tree if the keys are continuous. Note that this does not necessarily

rate balanced search trees over Braun Trees. The constants that are not measured by the complexity can be very different. Section 5.5 that provides measurements of all map implementations shows that an `BraunTree` implementation consumes less time and memory than the `Map` when it is used for the maps in the ROBDD implementation presented in this paper.

A `BraunTree` maps positive integer values to entries. We additionally need a mapping from tuples of integer values to entries. A map that uses tuple types as keys is implemented by a `BraunTree` whose entries are `BraunTree`s. This is a general construction that is used for Tries, too [18]. A map that uses a product of types as keys is implemented by an application of the maps for the component types. The measurements show that this implementation of maps for tuple types even has advantages when it is used for `Map`. Note that balanced search trees and therefore the `Map` implementation provide maps for arbitrary comparable key types.

There are two improvements for the `BraunTree` implementation. The first one is to use an finite data structure instead of an infinite one. We add an constructor `Empty` and use this one to represent the empty map. We call this data structure `BraunMap`. This implementation has advantages when we look up a key in an empty map. The key is not evaluated by this look-up. The look-up instantly yields `Nothing`. If we look up a key in an empty `BraunTree` we always have to evaluate the path to the entry we are looking for. In a `BraunMap` we can stop when we reach an `Empty` constructor.

Another improvement is using unboxing [27]. The `BraunTree` implementation needs a lot of memory. Every operation on an `Int` assigns a new memory cell in the heap. The Unboxing of the `Int` values prevents this. This feature is not part of the Haskell 98 Standard [20].

5.4 IntMap

The GHC supports an implementation of Patricia Trees [26]. A Patricia Tree is a Trie [25] for binary numbers that are represented by the corresponding integer value. If the keys that are inserted are not continuous a Trie can be very sparse. In a Patricia Tree empty parts of the Trie are merged. Patricia Trees are very similar to Braun Trees. The main difference is that the Braun Tree does not merge empty parts. We can emulate the behavior of one by the other. We present both data structures in this work because the `BraunMap` with unboxing is best when we use no optimizations and the `IntMap` comes with the GHC and is best when using optimizations.

The `IntMap` implementation uses an `UNPACK` pragma [20]. This pragma prompts the compiler to unbox the value behind the pragma. The values are reboxed if it is necessary for examples if the value is passed to a non-strict function. This pragma is used in combination with optimizations. Because of unfolding some of the reboxing is not needed. We define a map for tuples types like in the `BraunTree` implementation, i.e., by an `IntMap` that contains `IntMap`s.

5.5 Measurements

Figure 5.1 shows measurements of several map implementations. We only compare measurements for an implementation that uses a relaxed *no-redundancy* because this paper focuses on this implementation. We measure the construction together with an application of `eval` or `anySat` for two expressions. These applications use the *apply* and the *reverse map* but not the *memo map*. The *apply map* and the *reverse map* make the same demands on the map implementation. We have proved this by additional measurements where we changed the implementations independently.

It is unlikely for balanced search tree implementations to work well for one expression and bad for another. This is not necessarily true for a Braun or Patricia Tree implementation. The look-up and insert in a Braun and a Patricia Tree are logarithmic in the size of the key. The keys of the *apply* and the *reverse map* are tuples of `NodeId`s. If the ROBDD contains many nodes with one successor that is a reference to a node with a big `NodeId` many of the keys of the *reverse map* are big. That is, the look-up and insert of these keys is expensive. Therefore the efficiency of these implementations depends on the structure of the ROBDD. Hashing could be used to avoid this problem.

We start with the most common solution, the `Map`. The `Map2` implementation uses this balanced search tree implementation, too. Instead of using the map for tuple types that is supported by `Map` it uses a `BraunTree`-like implementation for tuples types, i.e., a `Map` that uses integer values as keys that contains `Map`s. This implementation is better than the standard implementation. When using optimizations the running times are half the ones for the standard `Map` implementation. Without optimizations the differences are not that significant.

The standard implementation searches a key by comparing the key tuple with the tuples in the map. The `Map2` implementation always compares only one component of the tuple. In the outermost map the first component of the tuple is looked up. If an entry is yielded the first component is discarded and the rest of the tuple is looked up. We never compare the first component again. In the `Map` implementation we have to compare the whole tuple. It would be interesting to check whether this implementation is always better than the one provided by the GHC and if this effect scales with the size of the tuple but this is out of the range of this thesis.

The `FiniteMap` implementation that uses no strictness annotations is nearly as good as the `Map`. The differences are more significant if we use optimizations.

The `BraunTree` implementation is better than the `Map2` implementation. Unboxing even improves this implementation because we do not assign a new memory cell for every integer operation. When we apply `anySat` to an ROBDD the `BraunTree` implementation requires more memory than the `BraunMap` implementation. This is also reflected in the number of evaluated constructors. While the `anySat` application to the ROBDD for the `Integer 16` expression evaluates 278560 with the `BraunTree` implementation it evaluates only 236 with every other implementation. In the `Queens 7` measurement it is 571235 evaluated constructors against 508083. The empty `BraunTree` is an infinite data structure whose entries are all `Nothing`. Thus, a look-up in an empty `BraunTree` causes the evaluation of the key that is looked up. All other implementations use a special

constructor for the empty map. Therefore these implementations yield `Nothing` without evaluating the key. The evaluation of the `NodeId` causes the evaluation of additional constructors in the ROBDD. Like predicted, the number of evaluated constructors is the same for all the other measurements.

Last we measure the `IntMap` implementation that comes with the GHC. Because the unbox pragma of the `IntMap` implementation has no effect without optimizations it cannot compete with the implementation of Braun Trees that uses explicit unboxing. If we use optimizations the `IntMap` is significantly better than the Braun Tree implementation with unboxing.

Map	Expression	Operation	Time	Memory
Map	Queens 7	anySat	18.12	2,518,950,244
		eval	22.68	3,152,943,608
	Integer 17	anySat	0.00	210,540
		eval	38.50	5,815,872,736
Map2	Queens 7	anySat	14.58	2,391,872,640
		eval	18.16	2,963,585,696
	Integer 17	anySat	0.00	212,088
		eval	24.24	4,134,822,020
FiniteMap	Queens 7	anySat	18.94	2,813,998,312
		eval	23.90	3,520,657,404
	Integer 17	anySat	0.00	206,500
		eval	40.48	6,544,851,040
BraunTree	Queens 7	anySat	15.32	1,979,628,088
		eval	17.00	2,213,574,228
	Integer 17	anySat	9.00	1,177,283,428
		eval	18.96	2,513,029,316
BraunMap	Queens 7	anySat	16.90	1,948,148,232
		eval	21.68	2,498,971,256
	Integer 17	anySat	0.00	210,116
		eval	24.50	2,901,153,896
BraunMap (unboxed)	Queens 7	anySat	5.64	703,744,940
		eval	7.26	880,183,040
	Integer 17	anySat	0.00	206,140
		eval	8.72	1,026,640,864
IntMap	Queens 7	anySat	17.30	2,946,999,464
		eval	21.98	3,749,171,680
	Integer 17	anySat	0.00	213,780
		eval	25.96	4,689,306,920

Figure 5.1: Measurements for several map implementations

The complexity of the look-ups and inserts in a Braun or a Patricia Tree are logarithmic in the size of the key while these are logarithmic in the number of elements in

the map for the `Map` implementation. For a Braun or a Patricia Tree these operations are logarithmic in the greatest component of the tuple for a map over tuples of integer values. For a balanced search tree these operations are still logarithmic in the number of elements in the map. The two implementations have equal complexities only if the keys are continuous. This is not the case for both, the *reverse* and the *apply map*. Although the complexities of the Braun and Patricia Tree implementations are worse then the complexities of balanced search trees they perform better in the measurements. This can have two reason. First the constant of the complexity of the balanced search tree implementations can be significantly greater than the one for the other implementations. Second the laziness of the Braun and Patricia Tree implementation can cause this difference.

All these measurements do not make any use of the *memo map*. Therefore we measure the use of this map by the application of `satCount`. Measurements of this application with a Braun Tree implementation for the *reverse* and the *apply map* showed that this map is also the best implementation for the *memo map*. The keys of the *memo map* are continuous. Therefore the complexity of look-up and insert of the Braun Tree implementations is the same as the complexity of these operations of the balanced search trees. The Braun Tree implementations even performed better with non continuous keys.

6 Related Work

6.1 Functional Implementations

There is only one ROBDD implementation in a functional language available [6]. Like stated on their page this implementation is a alpha version and not very efficient. This implementation is not likely to be improved since the page states "It is intended to improve the implementation throughout the coming year (97-98) and I will post any refinements and extra documentation as and when I write them.".

Instead of using a *map* and a *reverse map* this implementation uses only one list of tuples for both. The first component of each tuple is the `NodeId` of a node and the second is a triple consisting of its variable number and the `NodeId`s of its low and its high successor. This is not very efficient since a look-up in this map is linear in the number of nodes. The look-ups of the implementations presented in this paper are logarithmic in the number of nodes.

We compare the purely functional implementation using the relaxed *no-redundancy* property with the implementation of Jeremy Bradley. Figure 6.1 shows the results. The Bradley implementation cannot compete with the implementation presented here.

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 6.1: Comparison with the Bradley implementation

This implementation is far better no matter if we use a relaxed or a full *no-redundancy* property. The differences in the running times 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. However the memory usage of the Bradley implementation is much worse.

6.2 Functional Bindings to Imperative Implementations

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 [11]. This implementation defines an abstract interface to boolean expressions similar to the one presented in Section 4.5. 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 [12]. 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 [13] a model checker for the logic of knowledge written in Haskell. This implementation uses the *don't care* idea in the implementation of the top level boolean operators. That is, the boolean operators that are provided by the interface. For example the boolean operator `(∧)` instantly yields `false` if one of its arguments is `false`.

```
(∧) :: Boolean b => b → b → b
x ∧ y
   | x == false = false
   | x == true  = y
   | y == false = false
   | y == true  = x
   | otherwise  = x `bAND` y
```

The function `bAnd` is the Haskell binding of the function `apply` with the boolean operator $\wedge$ in the BDD Package. Therefore the construction of the ROBDD for the boolean expression $false \wedge exp$ does not evaluate exp even if we use HBDD. Note that the *don't care* idea saves the evaluation of ROBDDs only for the top level boolean operators, i.e., for an application to two ROBDDs and not for the applications on their sub-ROBDDs. The sub-ROBDDs are processed by the C implementation. This implementation can only save the traversal of the ROBDD but not its evaluation.

We compare our implementation with the HBDD binding. This is the more current binding to a BDD Package. We use optimizations in this measurements. The C implementation is highly optimized and therefore we want to bring out the best in our implementation. Besides we do not know the internal implementation of the BDD Package. Therefore we are unable to explain the differences in the measurements considering the internals anyway.

The HBDD implementation consumes less time and more memory than the implementation presented here in all the measurements. On the one hand the C implementation uses lots of refinements. A major one is the variable reordering. The best example for

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
Integer 1800	anySat	Relaxed	4.96	920,804,952
		HBDD	0,06	6,071,980
	eval	Relaxed		out of memory
		HBDD	0,04	4,049,024
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
HWB 13	anySat	Relaxed	1.38	381,347,160
		HBDD	0,70	87,341,268
	eval	Relaxed	2.64	587,033,824
		HBDD	0,68	87,323,952
HWB 17	anySat	Relaxed	29.92	8,431,064,836
		HBDD	13,34	1,792,605,492
	eval	Relaxed		out of memory
		HBDD		out of memory

Figure 6.2: Comparison with the HBDD implementation

the use of variable reordering is the `Integer` expression. This expression is of exponential size with the canonical variable order that is used by the implementation in this paper. 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 a low level language in comparison to Haskell and is therefore more efficient.

Our implementation gets closest to HBDD in the measurements of the `HWB` expression. The ROBDD size of `HWB` is exponential independent of the variable order. The differences in the sizes of the ROBDD for different variable orderings are very small [5]. Therefore the reordering of the variables is of little use. We cannot explain the out of memory exception that is caused by the application of `eval` on the ROBDD in the HBDD implementation. The compiler reports an out of memory exception for all applications of `eval` to the `HWB` expression with more than 13 variables although beforehand the correct result is displayed. Our implementation does not yield the result before it runs out of memory.

7 Summary

This paper presents a purely functional implementation of ROBDDs in Haskell. The motivation of this implementation is the use of lazy evaluation to prevent unnecessary computations in the construction of an ROBDD. To get any lazy behavior at all we relax the *no-redundancy* property of the ROBDDs. Besides the ROBDD data structure this paper presents the implementation of the most important operations on these ROBDDs.

7.1 Conclusion

This thesis demonstrates that data structures can benefit from laziness. We improve the performance of the operations on ROBDDs in respect of time and memory consumption. The ROBDD data structure is highly optimized, i.e., it contains few redundancies and contains many dependencies. This complicates the use of laziness. We have to relax the *no-redundancy* property to gain any laziness in the construction.

The relaxing of 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. This paper does not only present the implementation of the ROBDD data structure and investigates its laziness, it additionally proposes a modification of this data structure, shows its implementation and investigates the laziness of this modification.

The *no-redundancy* property of ROBDDs causes the evaluation of the whole ROBDD when we apply an operation to it. The relaxing of the *no-redundancy* property is an adequate answer to this problem. The disadvantages of the relaxing are small or not existent. The number of redundant nodes is very small for all examples we have measured. All operations more or less benefit from the laziness.

Even though the representation with relaxed *no-redundancy* is no more canonical the implementation of the equality check shows surprisingly good results. In the run-up of the implementation we expected this to be the most difficult function because it makes use of the canonical form. The implementation of the equality check for ROBDDs with relaxed *no-redundancy* property using `apply` has a quadratic worst case complexity in a strict language. In a lazy programming language we expect this operation to have a better complexity. There is only one paper about a lazy complexity of an algorithm [4]. This is a broad field of undiscovered research. For a fundamental investigation of the effect of lazy evaluation on the complexity of an algorithm the ROBDD data structure is too complex.

Apart from the benefits in the semantics it is very hard to benefit from laziness if the benefits 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. The available tools even abstract from lazy evaluation. The development of tools that support the design of lazy algorithms is another field of research that is not discovered yet.

The implementation presented here is far more efficient than the only known functional implementation. Even without the use of laziness this implementation beats the existing one. The Haskell implementation presented here cannot compete with a binding to an BDD Package implemented in C or C++. This comparison is not completely fair because these implementations use many refinements that are not used in the implementation presented here. One example is the variable reordering. Apart from these refinements an highly optimized implementation in a low level language like C is more efficient.

There are some disadvantages of the implementation with relaxed *no-redundancy*. A common extension of the ROBDD implementation to improve it is the use of one *reverse map* for all ROBDDs. The equality check of this implementation is in $O(1)$. We just have to compare the `NodeId`s of the root nodes of the two ROBDDs. Additionally we can use one *apply map* for all applications of `apply`. This increases the probability that application is memoized. This extension does not cooperate with laziness. When we look up the first node in an ROBDD in the *reverse map* we have to check whether we have already constructed this node. Therefore we have to evaluate the keys of the delayed inserts. That is, we have to evaluate the `NodeId`s of all nodes that were constructed so far. This causes the evaluation of all ROBDDs constructed so far.

A shortcoming of this paper is the lack of heap profiling, i.e., the observation how much heap memory is occupied at one moment. All the measurements in this paper only observe the overall memory usage. An observation that is highly related to this one is the execution of the garbage collector. There are measurements where the implementation with relaxed *no-redundancy* runs out of heap memory while the implementation with full *no-redundancy* does not although the relaxed implementation consumes less heap memory. Laziness often prevents data structures from being garbage collected. This is a disadvantage for the ROBDD construction because we can run out of memory only because there are parts of an ROBDD that cannot be garbage collected because of lazy evaluation.

7.2 Future Work

In the conclusion section some of the future work was already presented. There is one point that is rather future research than future work, namely a closer investigation of the lazy complexities of the presented algorithms. A future work is the implementation of the extension that uses one *reverse map* for all ROBDDs and the comparison of this implementation with the implementation that is presented in this paper. Another point where we have to do some work is the profiling of the heap memory.

We do not provide a good estimation of the number of redundant nodes in an ROBDD

with relaxed *no-redundancy* property. If we could support a closer estimation it would be easier to rate the disadvantages of the relaxing of the *no-redundancy* property. However even the operations on an ROBDD with full *no-redundancy* are not efficient in the worst case. That is, they only perform well in practice. The worst case number of redundant nodes is not required to be of any practical relevance. Therefore more measurements would probably be more significant.

Another extension that was proposed shortly after the introduction of the ROBDD data structure are Complement Edges, i.e., edges that complement all terminals of an sub-ROBDD. In the `OBDD` data type this can be implemented by an additional unary constructor `Not`. The most important advantage of this extension is that the negation of an ROBDD is in $O(1)$ by complementing the edge that leads to the root node. We have implemented this idea. The implementation only requires some minor changes mainly in the implementation of the functions `getLow` and `getHigh`. We have not measured this extension but some first tests showed that this extension cooperates with laziness.

Many modifications of ROBDDs have been published. These are motivated by the fact that the ROBDD representations of some functions that are used in practice are still exponentially large. Here is a list of some of the modifications and this list is far from complete: OFDDs, OKFDDs, parity OBDDs, FBDDs, BEDs, Partitioned-ROBDDs, Free BDDs, IBDDs, TBDDs, MTBDDs, ADDs. Rolf Drechsler and Detlef Sieling give a short introduction to the ideas of some of these extensions [28]. We have to relax the *no-redundancy* property of the ROBDDs to gain any laziness. It would be interesting whether any of these modifications would benefit from laziness without restricting their properties.

We hope that this thesis is only the start point of more research on the benefits and disadvantages of lazy evaluation for the efficiency of algorithms. Many experts advice to use strictness annotation in the definition of a data structure to improve the performance of the algoritms for this data structure. Still today eight years after the definition of the Haskell 98 Standard this issue is highly up-to-date like shown by some discussions on the mailing list to the new Haskell standard Haskell' [17].

We thank you for your attention!

Bibliography

[1] Adams, S., *Efficient sets: a balancing act*, in: *Journal of Functional Programming*, 1993, pp. 553–562.

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

[3] Andersen, H. R., *An introduction to binary decision diagrams* (1997), `http://www.itu.dk/people/hra/bdd97-abstract.html`.

[4] Bird, R., G. Jones and O. D. Moor, *More haste, less speed: lazy versus eager evaluation*, J. Funct. Program. **7** (1997), pp. 541–547.

[5] Bollig, B., M. Lobbing, M. Sauerhoff and I. Wegener, *On the complexity of the hidden weighted bit function for various BD D models*, Informatique Theorique et Applications **33** (1999), pp. 103–116.

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

[7] Braun, W. and M. Rem, *A logarithmic implementation of flexible arrays*, Memorandum MR83/4 (1983).

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

[9] Bryant, R. E., *On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication*, IEEE Trans. Comput. **40** (1991), pp. 205–213.

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

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

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

[13] 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.

[14] *The Glasgow Haskell compiler*, `http://www.haskell.org/ghc/`.

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

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

[17] *The haskell-prime archives* (2006), `http://www.haskell.org/pipermail/haskell-prime`.

[18] Hinze, R., *Generalizing generalized tries*, Journal of Functional Programming **10** (2000), pp. 327–351.

[19] 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`.

[20] Jones, S. P. et al., *Haskell 98 - a non-strict, purely functional language*, 1999.

[21] Launchbury, J., *A natural semantics for lazy evaluation*, in: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993, pp. 144–154.

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

[23] Long, *CMU BDD library* (1993), `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

[24] Nievergelt, J. and E. Reingold, *Binary search trees of bounded balance*, in: *SIAM journal of computing*, 1973.

[25] Okasaki, C., "Purely Functional Data Structures," Cambridge University Press, Cambridge, 1998.

[26] Okasaki, C. and A. Gill, *Fast mergeable integer maps*, in: *Workshop on ML*, 1998, pp. 77–86.

[27] Peyton Jones, S. L. and J. Launchbury, *Unboxed values as first class citizens in a non-strict functional language*, in: J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture* (1991), pp. 636–666.

[28] Rolf Drechsler, D. S., *Binary decision diagrams in theory and practice*, International Journal on Software Tools for Technology Transfer (STTT) **3** (2001), pp. 112–136.

[29] Sansom, P. M. and S. L. Peyton Jones, *Time and space profiling for non-strict higher-order functional languages*, in: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, 1995, pp. 355–366.

[30] Somenzi, F., *CU decision diagram package release 2.4.1* (1998), `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[31] *Satisfiability suggested format*, `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`.