Lean Binary Decision Diagrams

MSc Thesis *(Afstudeerscriptie)*

written by

Eshel S. Yaron

under the supervision of **Dr Malvin Gattinger**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
June 30th, 2025	Dr Benno van den Berg (Chair)
	Dr Malvin Gattinger (Supervisor)
	Dr Gregor Behnke
	Dr Andrés Goens

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

We develop a formally verified Binary Decision Diagram library in, and for, the Lean 4 programming language and proof assistant.

Binary Decision Diagrams (BDDs) are a fundamental data structure for efficient representation and manipulation of Boolean functions. Popularized by Bryant in 1986, BDDs have been implemented and deployed in various applications of computer science and logic, including circuit design and verification, model checking, planning and constraint solving.

The significance of our contribution of a formally verified BDD library in Lean is threefold: first, spelling out subtle correctness proofs for key BDD algorithms in a fully formal setting elucidates the essential logical properties of the various interacting components in these algorithms. As Knuth (2009) remarks after describing his BDD *reduction* algorithm, "the intricate link manipulations of Algorithm R are easier to program than to explain."

Additionally, the project has a practical benefit: a verified BDD implementation in Lean enables *proof automation* by leveraging Lean's *meta-programming* facility. Namely, proofs of various statements about Boolean functions, such as satisfiability of a given propositional formula or semantic equivalence of two formulae, can be delegated to a verified, efficient, BDD-based decision procedure within Lean.

Lastly, a verified BDD library can be used as a component of verified BDD-based applications, such as verified model checkers whose correctness is formally proven in Lean. We implement and present a BDD-based SAT solver as an example application of our library.

We describe the library's external API and its capabilities, explain the underlying implementation, discuss the design choices taken during formalization, and investigate implications of these choices. Finally we evaluate the performance of the presented implementation, compare it with other BDD implementations, and survey possibilities for further development.

Acknowledgements

I would like to thank my thesis advisor, Malvin, for outstanding guidance, encouragement and assistance during my work on this thesis. Our weekly meetings were extremely helpful, interesting and enjoyable. I couldn't have asked for a better supervisor, thank you!

I also want to thank my thesis committee members Gregor and Andrés, and the committee chair Benno, for reviewing my thesis and providing invaluable feedback in the thesis defense.

I'm grateful for the ILLC and the Master of Logic program in general. I was fortunate to meet incredibly welcoming and inspiring people during the past two years, and I gained and learned far more than I excepted. In particular, I want to thank Maria Aloni, who has been my academic mentor during the Master of Logic. Thank you, Maria, your mentorship and support were highly appreciated.

Another big thank-you goes to my friends, old and new, that supported me in this journey and enriched my experience with many fun moments.

Lastly, I want to thank my family for their endless care, support, and patience in hearing me ramble about logic. My greatest gratitude goes to my wife, Adi. You motivated and encouraged me every step of the way, and your belief in me means the world.

Contents

Chapter 1

Introduction

In his 1986 paper "Graph-Based Algorithms for Boolean Function Manipulation" [Bry86], Randal Bryant introduced a data structure for efficient representation of Boolean functions called ROBDD—Reduced Ordered Binary Decision Diagram. ROBDDs, or just BDDs for short, represent Boolean functions as directed acyclic graphs with certain structural constraints. They key benefits of BDDs are:

- *Compact*: Many important Boolean functions can be represented by BDDs using less space than with alternative representations.
- *Canonical*: There is a unique ROBDD corresponding to each Boolean function.
- *Efficient*: Common operations on Boolean functions can be implemented as efficient graph algorithms on BDDs.

Thanks to these benefits, BDDs have been implemented and applied to tackle various challenges in computer science, electrical engineering, and logic, including applications in CAD [Bry95], planning [BS21] and model checking [Gat18]. More than two decades after their introduction by Bryant, in 2008, Donald Knuth said about BDD that "it is one of the only really fundamental data structures that came out in the last twenty fives years." [Onl19]

In this thesis, we present the implementation of a formally verified BDD library in Lean 4—a functional programming language and formal proof environment based on Dependent Type Theory[MU21].

As a programming language, Lean 4 provides a rich functional programming environment with features such as pattern matching, type inference, and monadic programming constructs, while simultaneously serving as a powerful theorem prover with tactics for interactive proof development. This dual nature makes Lean 4 particularly well-suited for our purposes, as we can implement efficient BDD algorithms using familiar programming idioms, and formally prove their correctness within the same system. Additionally, Lean programs are compiled to efficient executable code, which allows our library to handle large BDDs with reasonable performance.

Our main references with regards to BDD implementation are Bryant's original paper from 1986 [Bry86] and his subsequent work on BDD implementation techniques along with other authors in the 1990's [BRB91; MIY91; Rud93; Bur+94; Bry95], as well as Knuth's writings about BDD in The Art of Computer Programming [Knu09], and Bryant's more recent overview in the Hanbook of Model Checking [Bry18]. We also take inspiration from existing implementations, most notably CUDD [Som98] and CacBDD [LSX13], which are both high-performance implementations written in low-level programming languages, along with implementations in higher-level languages such as Markus Triska's BDD implementation in Prolog [Tri16] and Masahiro Sakai's implementation in Haskell[1], as well as the formalization of BDDs in the Rocq theorem prover as described in [Ver+00].

In section 1.1, we discuss key notions and definitions of Boolean functions, which motivate the use of ROBDDs. In section 1.2, we introduce ROBDDs and show the main theoretical results that underpin their practical utility. In section 1.3, we examine Lean 4 and see some code examples.

In chapter 2, we present our Lean 4 ROBDD library. Section 2.1 describes the library's interface and usage, while section 2.2 goes into the details of our implementation and formalization of ROBDDs, including the formal correctness proofs for some of the fundamental ROBDD algorithms which our library implements. In section 2.3, we present and evaluate a verified SAT solver, as an example application of our library.

Lastly, in chapter 3 we discuss different directions for advancing our ROBDD library further.

1.1 Boolean Functions

We begin by examining and motivating the notion of Boolean functions. We can define Booleans in different ways depending on the formal settings we adopt. In this thesis, we shall mostly work within the context of Dependent Type Theory with Inductive Constructions, in which Booleans are defined as an inductive type (see section 1.3). Regardless of the technicalities, there is just one thing to know about Booleans: *there are exactly two distinct Booleans.* They are often called *true* and *false*, or sometimes 0 and 1. They represent the most basic kind of distinction, either *this* or *that*. If it is *this* then it is not *that*, and vice-versa. We could also call them "heads" and "tails", or "here" and "there". In this thesis we stick with *true* and *false*, and define the set of Booleans $\mathbb{B}$ as $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$.

Since a Boolean embodies a single *distinction*, a Boolean-valued *function* can be thought of as a single *decision*—based on some inputs, the function decides between "yes" and "no", *true* and *false*. Which inputs should we consider, when we think about functions that implement a single decision? It is natural to make decisions based on existing discernible distinctions. To decide whether to make an omelette, for example, one may distinguish between the case in which they have some eggs handy, and the case they have none.

In this example we can consider the input of the decision function to be a Boolean too, just like the output. In fact, we may need more than one Boolean to decide about the omelette, we may also need to know if the stove is working, for instance.

In general, we can model any procedure that makes a single decision based on finitely many distinctions as a function from Booleans to an output Booleans. These are called *Boolean functions.*

Definition 1 (Boolean Function)**.** *Given a natural number n, an n-ary* Boolean function *is a function $f : \mathbb{B}^n \to \mathbb{B}$. In words, f maps n input Booleans to one output Boolean.*

The number of Boolean functions grows rapidly with their arity: there are exactly 2 nullary Boolean functions, 4 unary functions, 16 binary functions, and 2^{2^n} n-ary functions. n-ary Boolean functions correspond to subsets of $\mathbb{B}^n$—each function f characterizes a set of *solutions* $S_f = \{x \mid f(x) = \mathsf{true}\}$.

Definition 2. *We fix the following notation:*

[1] https://github.com/msakai/haskell-decision-diagrams

- *We use* $\mathbf{0}$ *to refer to the constantly-*false *Boolean function* $x_0, \ldots, x_{n-1} \mapsto \mathsf{false}$*, and similarly* $\mathbf{1}$ *refers to* $x_0, \ldots, x_{n-1} \mapsto \mathsf{true}$*.*
- *We use* $\boldsymbol{x}_i$ *to refer to the "projection" function* $x_0, \ldots, x_{n-1} \mapsto x_i$ *that simply returns the value of the input at index* i*.*
- *For any Booleans* $b, b' \in \mathbb{B}$*:*
  - *The conjunction of* b, b' *is:*

$$b \cdot b' = \begin{cases} \mathsf{true} & \textit{if } b = b' = \mathsf{true} \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

  - *The disjunction of* b, b' *is:*

$$b + b' = \begin{cases} \mathsf{false} & \textit{if } b = b' = \mathsf{false} \\ \mathsf{true} & \textit{otherwise} \end{cases}$$

  - *The negation of* b *is:*

$$\bar{b} = \begin{cases} \mathsf{false} & \textit{if } b = \mathsf{true} \\ \mathsf{true} & \textit{otherwise} \end{cases}$$

- *We lift the above operation on Booleans to* n*-ary Boolean functions* f, g*: given an input vector* $x = x_0, \ldots, x_{n-1}$*, we have* $(f \cdot g)(x) = f(x) \cdot g(x)$*,* $(f + g)(x) = f(x) + g(x)$ *and* $\bar{f}(x) = \overline{f(x)}$*.*
- *For* $n \in \mathbb{N}$*, we use* $[n]$ *to refer to the set* $\{0, \ldots, n-1\}$*.*

We can *represent* a Boolean function in various ways. The minimal requirement for a representation of a Boolean function, or any function for that matter, is that it should allow us to mechanically determine the output of the function given (a suitable representation of) its input. The notation we defined in definition 2 is an example of such a representation—we can represent any Boolean function using expressions that consist of the basic functions $\mathbf{0}, \mathbf{1}, \boldsymbol{x}_i$ as well as conjunction, disjunction and negation—but perhaps the simplest representation of a Boolean function is the *truth table*, which explicitly lists the values of the function for any possible input. As an example, the following truth table represents the ternary majority function, which is true if and only if at least two of its inputs are true:

x_0	x_1	x_2	$f(x_0, x_1, x_2)$
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	false
false	true	true	true
false	true	false	false
false	false	true	false
false	false	false	false

Truth tables have the highly desirable property (which our algebraic notation lacks) that they provide a *canonical* representation, which means that two Boolean functions are the same (in the sense that they have the same extension—they return the same output when given the same inputs) if and only if their truth tables are the same. More precisely, given two truth tables, it is trivial to check whether they represent the same Boolean function—just sort both tables and compare line by line.

However, the canonicity of truth tables comes at the expense of *compactness*, another desirable property truth tables lack completely. The truth table of an n-ary Boolean function consists of 2^n lines—one for each possible input—making them maximally verbose.

As an extreme example of this verbosity we can consider the truth table of the constantly-true ternary Boolean function **1**, which disregards its input completely and always returns true. Intuitively, this function is as simple as can be, certainly simpler than the aforementioned majority function. However, the truth tables of both functions are just as large:

x_0	x_1	x_2	$f(x_0, x_1, x_2)$
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	true
false	true	true	true
false	true	false	true
false	false	true	true
false	false	false	true

Since there are 2^{2^n} n-ary Boolean functions, any representation scheme will yield exponentially large representations for some functions. In other words, no representation is absolutely compact. However, some representations, including Binary Decision Diagrams (which we introduce in the next section), require less space to represent *some* functions. In particular, all symmetric functions admit very small Binary Decision Diagrams. (See [Knu09] for a detailed analysis of "BDD-friendly" Boolean functions.) Since some Boolean functions occur more commonly than others in many settings of interest, such partial compactness can provide great practical benefits.

Beyond merely providing a more compact representation, Binary Decision Diagrams (BDDs) also admit efficient algorithms for answering questions about the Boolean functions that they represent. In [Knu09], Knuth enumerates various such "BDD virtues", here we focus on the following:

Given a BDD representing a Boolean function of n variables f,

1. we can efficiently *evaluate* $f(x_0, \ldots, x_{n-1})$ for given inputs $x_0, \ldots, x_{n-1}$;

2. we can *check whether f is constant* (either constantly-true or constantly-false) in constant time, whereas answering the same question given the truth table of f requires checking up to 2^n entries;

3. more generally, we can *check whether two functions are equal* given their BDDs in time that is proportional to the size of the smaller of the two BDDs, which can be exponentially smaller than the corresponding truth table;

4. we can *find* inputs $x_0, \ldots, x_{n-1}$ such that $f(x_0, \ldots, x_{n-1}) =$ true (and likewise for false) in $O(n)$

time, instead of the exponential time we would need to find such inputs given the truth table of f, or given f as an "oracle";

5. we can efficiently *check whether f depends on* a given variable index (see definition 4 below), or find all indices the function depends on.

In preparation for discussing Binary Decision Diagrams in more detail, we introduce two more important definitions about Boolean functions: their *restrictions* and *Shannon expansions*.

Definition 3 (Restriction). *The* restriction *of an n-ary Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ at index $i \in [n]$ to value $b \in \mathbb{B}$, denoted $f_{i \leftarrow b}$, is the function $x_0, \ldots, x_{n-1} \mapsto f(x_0, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_{n-1})$. In other words, it is the same as f except that the input at index i is ignored and unconditionally replaced with the Boolean b.*

Definition 4 (Dependency on Input Variable). *We say that a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$* depends on *variable index i if $f_{i \leftarrow \mathsf{false}} \neq f_{i \leftarrow \mathsf{true}}$. In other words, there exist $x_0, \ldots, x_{i-1}$ and $x_{i+1}, \ldots, x_{n-1}$ such that*

$$f(x_0, \ldots, x_{i-1}, \mathsf{false}, x_{i+1}, \ldots, x_{n-1}) \neq f(x_0, \ldots, x_{i-1}, \mathsf{true}, x_{i+1}, \ldots, x_{n-1})$$

Since the restriction of an n-ary Boolean function "ignores" one of its inputs, we can think of it as an $n-1$-ary function. Formally:

Lemma 5. *For an n-ary Boolean function f, for any restriction $f_{i \leftarrow b}$ of f, there is an $n-1$-ary function g such that $g(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}) = f_{i \leftarrow b}(x_0, \ldots, x_{n-1})$ for all $x_0, \ldots, x_{n-1}$.*

Proof. We set $g = y_0, \ldots, y_{n-2} \mapsto f(y_0, \ldots, y_{i-1}, b, y_i, \ldots, y_{n-2})$, and get that for all $x_0, \ldots, x_{n-1}$, $g(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}) = f_{i \leftarrow b}(x_0, \ldots, x_{n-1})$, as needed. □

Definition 6 (Shannon Expansion). *The* Shannon expansion *of a non-nullary Boolean function f around input variable i is the function given by $\boldsymbol{x}_i \cdot f_{i \leftarrow \mathsf{true}} + \bar{\boldsymbol{x}}_i \cdot f_{i \leftarrow \mathsf{false}}$ [Sha38].*

Fact 7. *Boolean functions are extensionally equal to their Shannon expansions.*

The Shannon expansion allows us to express an n-ary Boolean function in terms of simpler, $n-1$-ary Boolean functions (its restrictions)—a process we can repeat all the way down to the nullary constant Boolean functions. Intuitively, deconstructing a Boolean function f along its Shannon expansion can be understood as representing f as a series of "if-then-else" case distinctions, corresponding to the input variables which we expand f around. This intuition underlies the use of Binary Decision Diagrams to represent Boolean functions.

1.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a data structure introduced in [Bry86] that provides a symbolic representation of Boolean functions. [CG18] gives a detailed historical perspective about the development and use of BDDs, in particular from the point of view of the use of BDDs in applications related to model checking. In this section we introduce the basic theory of BDDs and the properties that make them such a useful representation, before discussing our concrete BDD implementation in Lean in chapter 2.

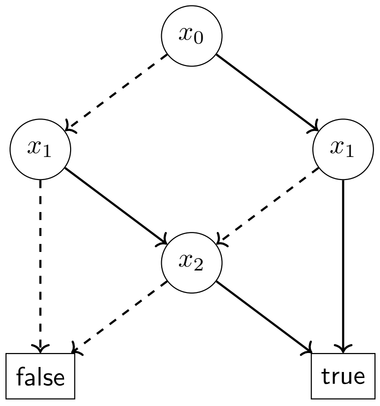

Figure 1.1: Binary Decision Diagram denoting the ternary majority function

Definition 8 (Binary Decision Diagram). *Given a natural number $n \in \mathbb{N}$, a* Binary Decision Diagram *of n variables is a tuple $\langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle$, where $\langle V, E\rangle$ is a finite directed acyclic graph,* $\mathrm{root} \in V$ *is a node in the graph, and* $\mathrm{var}\colon V \to [n]$, $\mathrm{val}\colon V \to \mathbb{B}$ *and* $\mathrm{label}\colon E \to \mathbb{B}$ *are functions such that:*

- *All nodes $v \in V$ either have exactly two outgoing edges or none at all. We call nodes with no outgoing edges* terminals*, and we call those with two outgoing edges* nonterminals*.*
- *For a nonterminal $v \in V$ with outgoing edges e_1, e_2, we have* $\mathrm{label}(e_1) \neq \mathrm{label}(e_2)$*. In other words, one edge is labeled with* true *and the other is labeled with* false*—we call them the* high *and* low *edges, respectively.*

We denote the set of BDDs of n variables by BDD_n, or just BDD when n can be inferred from the context. We also call n the input size *of a BDD $B \in BDD_n$.*

Definition 9. *For a BDD $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$ and node $v \in V$, we write $B[v]$ for the BDD that is the same as B except with v as root, $B[v] = \langle V, E, v, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle$.*

For nonterminals nodes $v \in V$, we write $\mathrm{high}(v)$ *for the target node of node v's high edge, and* $\mathrm{low}(v)$ *for the target of its low edge.*

Furthermore, if root *is nonterminal, we use* $\mathrm{high}(B)$ *as an abbreviation for* $B[\mathrm{high}(\mathrm{root})]$*, and similarly* $\mathrm{low}(B)$ *abbreviates* $B[\mathrm{low}(\mathrm{root})]$*. Note that* $\mathrm{high}(B)$ *is a BDD, while* $\mathrm{high}(v)$ *is a node.*

We also use $\mathrm{var}(B)$ *as an abbreviation for* $\mathrm{var}(\mathrm{root})$ *when that is more convenient.*

We define the *size* of a BDD as follows:

Definition 10 (BDD size). *The* size *of a BDD B,* $\mathrm{size}(B)$*, is the number of nonterminal nodes reachable from its root,* $\mathrm{root}(B)$*.*

We only take into account nonterminal nodes when we consider the *size* of a BDD because although we didn't require it up front, in practice we can assume there are (at most) two terminal nodes (one for each Boolean value). In particular, this assumption always holds for *reduced* BDDs. In our implementation (chapter 2) we go a step further and represent terminal nodes implicitly, as distinguished pointers, so we do not allocate any memory for them.

We associate with each BDD node v a Boolean function, which we call the *denotation* of v or $d(v)$:

Definition 11 (BDD Denotation)**.** *For a BDD of n variables $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$, the denotation function $D_n \colon BDD_n \mapsto \mathbb{B}^n \to \mathbb{B}$ is given by:*

- *If* root *is a terminal node, then the denotation of B is the constant function*

$$D_n(B) = x_0, \ldots, x_{n-1} \mapsto \mathrm{val}(\mathrm{root})$$

- *If* root *is nonterminal, the denotation of B is*

$$D_n(B) = \boldsymbol{x}_{\mathrm{var(root)}} \cdot D_n(\mathrm{high}(B)) + \bar{\boldsymbol{x}}_{\mathrm{var(root)}} \cdot D(\mathrm{low}(B))$$

(Note that the graph's acyclicity guarantees that the denotation function D_n is well-defined.) We often omit the index n and write simply D when n is not pertinent or clear from the context.

The definition of the denotation of a nonterminal mirrors the Shannon expansion of Boolean functions: indeed, we have $D(B)_{\mathrm{var(root)}\leftarrow\mathsf{true}} = D(\mathrm{high}(B))$ and $D(B)_{\mathrm{var(root)}\leftarrow\mathsf{false}} = D(\mathrm{low}(B))$, and thus $D(B)$ is given by its Shannon expansion around the input variable var(root).

Lemma 12. *For all n-ary Boolean functions f there exists a BDD $B \in BDD_n$ such that $D(B) = f$.*

Proof. By induction on n.

Base If $n = 0$, then f is either $\mathbf{0}$ or $\mathbf{1}$. If $f = \mathbf{0}$, then any BDD B whose root is a terminal with $\mathrm{val}(\mathrm{root}(B)) = \mathsf{false}$ gives $D(B) = f$. If $f = \mathbf{1}$, then any BDD B whose root is a terminal with $\mathrm{val}(\mathrm{root}(B)) = \mathsf{true}$ gives $D(B) = f$.

Step If $n = k + 1$, then by fact 7 we get that $f = \boldsymbol{x}_k \cdot f_{k\leftarrow\mathsf{true}} + \bar{\boldsymbol{x}}_k \cdot f_{k\leftarrow\mathsf{false}}$. By lemma 5, there exist k-ary functions h, ℓ such that $h(x_0, \ldots, x_{k-1}) = f_{k\leftarrow\mathsf{true}}(x_0, \ldots, x_k)$ and $\ell(x_0, \ldots, x_{k-1}) = f_{k\leftarrow\mathsf{false}}(x_0, \ldots, x_k)$ for all $x_0, \ldots, x_k \in \mathbb{B}^{k+1}$.

Thus $f(x_0, \ldots, x_k) = \boldsymbol{x}_k \cdot h(x_0, \ldots, x_{k-1}) + \bar{\boldsymbol{x}}_k \cdot \ell(x_0, \ldots, x_{k-1})$.

By I.H., there exists BDDs $H, L \in BDD_k$ such that $D(H) = H$ and $D(L) = \ell$. Hence for a BDD B with $\mathrm{high}(B) = H$, $\mathrm{low}(B) = L$ and $\mathrm{var}(B) = k$ we get $D(B)(x_0, \ldots, x_k) = \boldsymbol{x}_k \cdot h(x_0, \ldots, x_{k-1}) + \bar{\boldsymbol{x}}_k \cdot \ell(x_0, \ldots, x_{k-1}) = f(x_0, \ldots, x_k)$, as needed. □

Lemma 12 shows that the BDD denotation function D is surjective: every Boolean function can be represented by a BDD. However, for arbitrary BDDs, the denotation function is not injective: we can represent the same Boolean function in multiple ways (for example, fig. 1.1 and fig. 1.3 show two different BDDs with the same denotation). We would very much like for the denotation function to be injective, because that would mean that our representation is canonical, and that we can check whether two functions are equal by comparing their concrete representations.

To remedy the situation, instead of considering all BDDs, we restrict our attention (and the domain of our denotation function) to *Ordered* BDDs, for which there exists a canonical form, called the *Reduced Ordered* BDD.

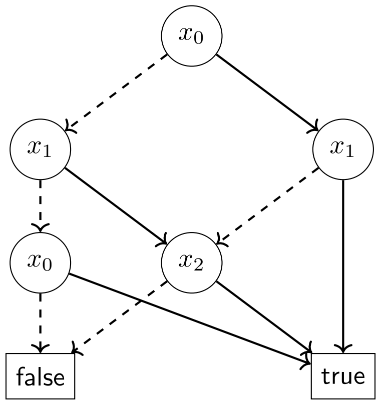

Figure 1.2: Example of a forgetful BDD

Definition 13 (Ordered BDD)**.** *A BDD $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$ is called* Ordered *if, for any edge $\langle v, u\rangle \in E$ reachable from* root *where u is a nonterminal, we have* $\mathrm{var}(v) < \mathrm{var}(u)$*. We abbreviate Ordered BDD to OBDD.*

In other words, a BDD is ordered if on all paths from its root down to the terminal nodes, the variable indices we encounter are in strictly increasing order.

While orderedness is a very natural property to consider, the BDD literature rarely clarifies its importance. Even in the original BDD paper [Bry86], orderedness is simply included as part of the definition of a BDD, without prior motivation. We suggest two motivations for restricting our attention to ordered BDDs:

1. Orderedness excludes an undesirable property that we call *forgetfulness*.
2. The number of nodes in ordered BDDs of n variables is upper bounded by 2^n, which is the size of the corresponding truth tables.

Definition 14 (Forgetful BDD)**.** *A BDD $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$ is called* Forgetful *if there exists a path from* root *that includes distinct nonterminals v, u such that* $\mathrm{var}(v) = \mathrm{var}(u)$*.*

Each path down from the root corresponds to a series of "checks", or distinctions, that the represented function may make in order to make a decision (by reaching a terminal node). A forgetful BDD encodes such a series in which we examine the value of the same variable twice, as if we forgot its value in between. This is, of course, an undesirable property that we would rather avoid. It is related to orderedness by the following fact:

Fact 15. *If a BDD $B \in BDD_n$ is* ordered up to variable permutation*, which means that there exists a permutation (bijective) function $\pi\colon [n] \hookrightarrow [n]$ such that replacing B's* var *function with the composition* $\pi \circ \mathrm{var}$ *yields an ordered BDD, then B is* not forgetful*.*

In addition, orderedness entails an upper bound on the size of the BDD:

Lemma 16. *For ordered BDDs $B \in BDD_n$ we have* $\mathrm{size}(B) \leq 2^n - 1$*.*

Proof. Since B is ordered, every variable index may appear at most once on any path from $\mathrm{root}(B)$, thus any such path includes at most n nonterminal nodes. Thus $\mathrm{size}(B)$ is bounded by the number of nodes in a full binary tree of height n, which is $2^n - 1$. □

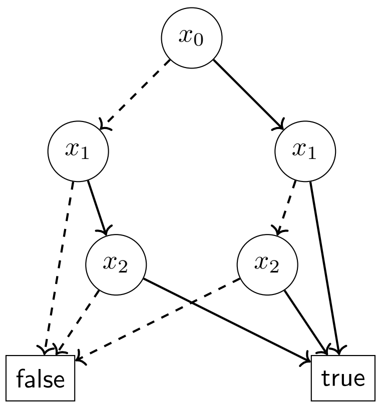

Figure 1.3: Another BDD denoting the ternary majority function

The following useful lemma establishes that all sub-BDDs of an ordered BDD are also ordered:

Lemma 17. *Given an ordered BDD $B \in BDD_n$, for all nodes v reachable from* $\mathrm{root}(B)$ *we have that $B[v]$ is also ordered.*

Proof. Let $\langle u, u' \rangle$ be an edge between two nonterminal nodes u, u' reachable from $\mathrm{root}(B[v]) = v$. Since v is reachable from $\mathrm{root}(B)$ and reachability is transitive, we get that u and u' are also reachable from $\mathrm{root}(B)$. Thus, by orderedness of B, we get that $\mathrm{var}(u) < \mathrm{var}(u')$. Thus $B[v]$ is ordered. □

Fact 15 and lemma 16 show that Ordered BDDs are relatively well behaved, but they are still not canonical with respect to Boolean functions. For example, fig. 1.1 and fig. 1.3 show distinct BDDs, which are both ordered and both denote the same function—the ternary majority function.

Fortunately, the *smallest* OBDD representing a given Boolean function *is* unique, and we can efficiently *reduce* any OBDD into this minimal form. These *reduced* OBDDs will act as our canonical representation for Boolean functions.

First, we define a notion of *Reduced* OBDDs as a subset of *BDD* satisfying certain structural constraints; then we show that Reduced OBDDs are canonical; and lastly we show that they are minimal in size.

Intuitively, an OBDD is reduced when it encodes a given Boolean function as succinctly as possible—it is the smallest OBDD for that function. Of course, a priori, there may be multiple distinct such minimal OBDDs for the same function; the canonicity theorem that we show below (theorem 25) ensures that reduced OBDDs representing a given function are unique (up to an equivalence relation which we introduce in definition 20), which justifies talking about reduced OBDDs as *the* most succinct OBDD with a given denotation.

Our definition of being reduced relies on a translation of BDDs to a simpler data structure that we call *decision trees*:

Definition 18 (Decision Tree)**.** *A* decision tree *is defined inductively as follows:*

$$t ::= b \mid i\,?\,t : t \qquad (\textit{where } b \in \mathbb{B}, i \in \mathbb{N})$$

A decision tree t of the form b for $b \in \mathbb{B}$ is called a leaf*, while decision trees of the form $i\,?\,t_1 : t_2$ are called* branches*. We read a branch $i\,?\,t_1 : t_2$ as an if-then-else instruction: if i then follow t_1, otherwise follow t_2.*

Definition 19 (Decision Tree of BDD)**.** *We associate with each BDD* $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$ *a decision tree* $T(B)$ *as follows:*

- *If* root *is a terminal node, then* $T(B) = \mathrm{val}(\mathrm{root})$.
- *Otherwise, if* root *is nonterminal, then* $T(B) = \mathrm{var}(\mathrm{root}) \mathbin{?} T(\mathrm{high}(B)) : T(\mathrm{low}(B))$.

Definition 20 (Similar BDDs)**.** *We say that two BDDs* B, B' *are* similar *and write* $B \sim B'$ *if* $T(B) = T(B')$.

For example, the BDDs shown in fig. 1.1 and fig. 1.3 are similar, since both of them induce the same decision tree $0 \mathbin{?} (1 \mathbin{?} \mathsf{true} : (2 \mathbin{?} \mathsf{true} : \mathsf{false})) : (1 \mathbin{?} (2 \mathbin{?} \mathsf{true} : \mathsf{false}) : \mathsf{false})$.

Lemma 21. *The BDD similarity relation* $\sim$ *is an equivalence relation.*

Proof. By definition, $\sim$ is the inverse image of the equivalence relation $=$ by the T function, hence it is also an equivalence relation. □

In essence, $T(B)$ is a binary tree that forgets the graph structure of B and encodes only the branching decisions that B prescribes. Note how the translation of a BDD B to decision trees T corresponds to the translation to Boolean functions D from definition 11 above—indeed, BDDs with the same decision tree also have the same denotation:

Lemma 22. *For BDDs* $B, B' \in BDD_n$, $B \sim B'$ *implies* $D(B) = D(B')$.

Proof. Suppose $B \sim B'$, thus $T(B) = T(B')$. By induction on $T(B)$:

- If $T(B)$ is a Boolean b, then $\mathrm{root}(B)$ is a terminal with value b, so $D(B)$ is the constantly-b function, and likewise for $D(B')$, since $T(B) = T(B')$.
- If $T(B) = i \mathbin{?} T_1 : T_0$, then $\mathrm{root}(B)$ is a nonterminal with variable index i, and we have $T_1 = T(\mathrm{high}(B))$ and $T_0 = T(\mathrm{low}(B))$. Since $T(B) = T(B')$, we get that $\mathrm{root}(B')$ is also a nonterminal with variable index i, and that $T(\mathrm{high}(B)) = T(\mathrm{high}(B'))$ and $T(\mathrm{low}(B)) = T(\mathrm{low}(B'))$, and so by I.H. we have $D(\mathrm{high}(B)) = D(\mathrm{high}(B'))$ and $D(\mathrm{low}(B)) = D(\mathrm{low}(B'))$.

  By definition of D, we get:

$$D(B) = \boldsymbol{x}_i \cdot D(\mathrm{high}(B)) + \bar{\boldsymbol{x}}_i \cdot D(\mathrm{low}(B)) = \boldsymbol{x}_i \cdot D(\mathrm{high}(B')) + \bar{\boldsymbol{x}}_i \cdot D(\mathrm{low}(B')) = D(B')$$

□

As a necessary condition for an OBDD B to be reduced, B must not contain distinct similar sub-OBDDs, otherwise if B had two distinct similar OBDDs, say S_1, S_2, then we could obtain a more succinct encoding of the same Boolean function that B denotes, by changing all edges that point to $\mathrm{root}(S_2)$ such that they point to $\mathrm{root}(S_1)$ instead, and dropping $\mathrm{root}(S_2)$ altogether. The resulting OBDD would still induce the same decision tree, and hence it would denote the same Boolean function, while its *size* would decrease.

However, lack of distinct similar sub-OBDDs is not sufficient for an OBDD to be reduced. Another way for an OBDD to be extraneously large is to include trivial decision nodes, wherein both the low and high edges lead to the same decision. Such a node increases the size of the OBDD, and provides nothing but frustration in return. It is akin to a asking a stranger for directions to the train station, and being asked in response if you prefer dogs or cats. Whether you are a dog person or a cat person, the way to the train station is the same, so the interchange is entirely redundant. We formalize this notion of redundancy in the following definition:

Definition 23 (Redundant BDD Node). *A BDD node v is* redundant *if it is a nonterminal with* $\mathrm{low}(v) = \mathrm{high}(v)$.

The notions of similarity and redundancy suffice for defining the notion of a *reduced* (O)BDD:

Definition 24 (Reduced BDD). *A BDD $B = \langle V, E, \mathrm{root}, \mathrm{var}, \mathrm{val}, \mathrm{label}\rangle \in BDD_n$ is* reduced *if the following two conditions hold:*

- *No redundant nodes are reachable from* root.
- *For all $v, u \in V$ that are reachable from* root, *$B[v] \sim B[u]$ implies $v = u$. In other words, there are no two distinct sub-BDDs of B that are similar.*

We can now state the fundamental theorem of BDDs, which establishes that Reduced Ordered BDDs (ROBDDs) are *canonical*:

Theorem 25 (Canonicity). *For Reduced and Ordered BDDs B, B', $D(B) = D(B')$ implies $B \sim B'$.*

Proof. Let $B, B' \in BDD_n$ be two ROBDDs with $D(B) = D(B')$. By strong induction on the sum of the sizes of B, B', we assume that the theorem holds for all pairs of BDDs C, C' such that $\mathrm{size}(C) + \mathrm{size}(C') < \mathrm{size}(B) + \mathrm{size}(B')$. We proceed by case distinction on the roots of B and B'.

- If $\mathrm{root}(B)$ is a terminal node with $\mathrm{val}(\mathrm{root}(B)) = b$ and $\mathrm{root}(B')$ is a terminal node with $\mathrm{val}(\mathrm{root}(B')) = b'$, then by definition of the denotation function D we have $D(B) = x \mapsto b$ ($D(B)$ is the constant function with value b), and $D(B') = x \mapsto b'$. By assumption, $x \mapsto b = D(B) = D(B') = x \mapsto b'$, and thus $b = b'$. Hence B, B' both induce the same decision tree, the leaf b, and are thus similar.
- If $\mathrm{root}(B)$ is a terminal node with $\mathrm{val}(\mathrm{root}(B)) = b$ and $\mathrm{root}(B')$ is nonterminal, then we have:

  $$x \mapsto b = D(B) = D(B') = \boldsymbol{x}_{\mathrm{var}(B')} \cdot D(\mathrm{high}(B)) + \bar{\boldsymbol{x}}_{\mathrm{var}(B')} \cdot D(\mathrm{low}(B))$$

  Hence $D(\mathrm{high}(B)) = x \mapsto b = D(\mathrm{low}(B))$. Since $\mathrm{size}(\mathrm{high}(B)) + \mathrm{size}(\mathrm{low}(B)) < \mathrm{size}(B) \leq \mathrm{size}(B) + \mathrm{size}(B')$, we can apply our induction hypothesis and get that $\mathrm{high}(B) \sim \mathrm{low}(B)$. Hence B is not reduced. Contradiction.

  The case in which $\mathrm{root}(B)$ is nonterminal and $\mathrm{root}(B')$ is terminal is analogous.
- If both $\mathrm{root}(B)$ and $\mathrm{root}(B')$ are nonterminals, then to show $T(B) = T(B')$ we first show that $\mathrm{var}(B) = \mathrm{var}(B')$.

  Suppose, for contradiction, that $\mathrm{var}(B) \neq \mathrm{var}(B')$. Without loss of generality, assume that $\mathrm{var}(B) < \mathrm{var}(B')$. Since B' is ordered, we get that the denotation of B' $D(B')$ is independent of $\mathrm{var}(B)$ (in our formalization in Lean, this implication is justified by lemma `independentOf_lt_root`—see section 2.2.4 where we discuss our formalization of this proof).

Since $D(B) = D(B')$, we get that $D(B)$ is also independent of $\mathrm{var}(B)$. But that implies that $D(\mathrm{high}(B)) = D(\mathrm{low}(B))$, and by I.H. we get that $\mathrm{high}(B) \sim \mathrm{low}(B)$ and hence B is not reduced. Contradiction. Thus $\mathrm{var}(B) = \mathrm{var}(B')$.

It remains to show that $T(\mathrm{high}(B)) = T(\mathrm{high}(B'))$ and that $T(\mathrm{low}(B)) = T(\mathrm{low}(B'))$. Since $D(B) = D(B')$ and $\mathrm{var}(B) = \mathrm{var}(B')$, we get that $D(\mathrm{high}(B)) = D(\mathrm{high}(B'))$ and $D(\mathrm{low}(B)) = D(\mathrm{low}(B'))$.

Since $\mathrm{size}(\mathrm{high}(B)) < \mathrm{size}(B)$ and $\mathrm{size}(\mathrm{high}(B')) < \mathrm{size}(B')$, we have $\mathrm{size}(\mathrm{high}(B)) + \mathrm{size}(\mathrm{high}(B')) < \mathrm{size}(B) + \mathrm{size}(B')$. Hence, by I.H., we get $T(\mathrm{high}(B)) = T(\mathrm{high}(B'))$, and similarly for $T(\mathrm{low}(B)) = T(\mathrm{low}(B'))$. □

Corollary 26. *For ROBDDs B, B', $D(B) = D(B')$ if and only $B \sim B'$.*

Proof. By theorem 25 and lemma 22. □

Corollary 27. *For an ROBDD B,*

- *$D(B) = \mathbf{0}$ if and only if $\mathrm{root}(B)$ is a terminal with $\mathrm{val}(\mathrm{root}(B)) = \mathsf{false}$.*
- *$D(B) = \mathbf{1}$ if and only if $\mathrm{root}(B)$ is a terminal with $\mathrm{val}(\mathrm{root}(B)) = \mathsf{true}$.*

Proof. If $\mathrm{root}(B)$ is a terminal with $\mathrm{val}(\mathrm{root}(B)) = \mathsf{false}$, then by definition of D we have $D(B) = \mathbf{0}$, and thus by canonicity we get for all ROBDDs B' such that $D(B) = \mathbf{0}$ we have $B \sim B'$, and thus B' is a terminal with $\mathrm{val}(\mathrm{root}(B')) = \mathsf{false}$.

The same holds for true instead of false. □

Theorem 25 tells us that if two ROBDDs encode the same Boolean function, then they are similar. Thus if we have two BDDs B_f, B_g representing n-ary Boolean functions f, g respectively, we can readily decide whether $f = g$ by checking if $B_f \sim B_g$, which takes $O(\min\{\mathrm{size}(B_f), \mathrm{size}(B_g)\})$ time using an algorithm that we present in section 2.2.4.

In particular, by taking $g \in \{\mathbf{0}, \mathbf{1}\}$, we get from corollary 27 that we can check whether f is a tautology or a contradiction in constant time. This result demonstrates the power of BDDs: given the BDD representing a Boolean function, problems such as satisfiability become trivial. Of course, satisfiability checking is notoriously NP-hard, which implies that constructing ROBDDs for arbitrary Boolean functions is also NP-hard.

Using our canonicity result, we can also prove that ROBDDs are the minimal OBDDs with a given denotation. First, we define the notion of minimality for BDDs, and prove a couple of helpful lemmas:

Definition 28 (Minimal BDD)**.** *A BDD $B \in BDD_n$ is called* minimal *if for all $B' \in BDD_n$, $D(B) = D(B')$ implies $\mathrm{size}(B) \leq \mathrm{size}(B')$.*

Lemma 29. *Given a reduced OBDD B, for all nodes v reachable from $\mathrm{root}(B)$ we have that $B[v]$ is also reduced.*

Proof. Similar to the proof of lemma 17. □

Lemma 30. *For two ROBDDs $R, R' \in BDD_n$, if $R \sim R'$ then $\mathrm{size}(R) = \mathrm{size}(R')$.*

Proof. If $\operatorname{root}(R)$ is a terminal, then $\operatorname{root}(R')$ must also be a terminal with $\operatorname{val}(\operatorname{root}(R)) = \operatorname{val}(\operatorname{root}(R'))$, and thus $\operatorname{size}(R) = 0 = \operatorname{size}(R')$.

Otherwise, we proceed by strong induction on the size of R. The induction hypothesis says that for all ROBDDs $B, B' \in BDD_n$ such that $\operatorname{size}(B) < \operatorname{size}(R)$, we have that $B \sim B'$ implies $\operatorname{size}(B) = \operatorname{size}(B')$.

Suppose that $\operatorname{size}(R) \neq \operatorname{size}(R')$. Without loss of generality, we assume that $\operatorname{size}(R) < \operatorname{size}(R')$. Hence more nodes are reachable from $\operatorname{root}(R')$ than from $\operatorname{root}(R)$.

Thus either $\operatorname{size}(\operatorname{low}(R)) < \operatorname{size}(\operatorname{low}(R'))$ (in words, there are more nodes reachable from $\operatorname{low}(R)$ than from $\operatorname{low}(R')$), or $\operatorname{size}(\operatorname{high}(R)) < \operatorname{size}(\operatorname{high}(R'))$. Again without loss of generality, we assume that $\operatorname{size}(\operatorname{low}(R)) < \operatorname{size}(\operatorname{low}(R'))$.

Since R and R' are ordered and reduced, by lemma 17 and lemma 29 we get that $\operatorname{low}(R)$ and $\operatorname{low}(R')$ are also ordered and reduced. Thus by induction hypothesis, we have that $\operatorname{low}(R) \nsim \operatorname{low}(R')$, and hence $R \nsim R'$. Contradiction.

Thus $\operatorname{size}(R) = \operatorname{size}(R')$, as needed. □

Theorem 31. *For a BDD $R \in BDD_n$, R is minimal if and only if R is reduced.*

Proof. We define a relation $\cdot \Rightarrow \cdot \subseteq BDD_n \times BDD_n$ which we call the "reduction step" relation, as the union of two relations $\Rightarrow = (\Rightarrow_1 \cup \Rightarrow_2)$, where:

- $\cdot \Rightarrow_1 \cdot \subseteq BDD_n \times BDD_n$, which we call the "redundancy elimination step" relation, is defined as: $B \Rightarrow_1 B'$ if and only if B can be obtained from B' by picking any node v reachable from $\operatorname{root}(B')$, adding a redundant node v' such that $\operatorname{low}(v') = \operatorname{high}(v') = v$, and changing all edges with target v to point to v' instead.

- $\cdot \Rightarrow_2 \cdot \subseteq BDD_n \times BDD_n$, which we call the "duplication elimination step" relation, is defined as: $B \Rightarrow_2 B'$ if and only if B can be obtained from B' by picking any node v that has multiple incoming edges reachable from $\operatorname{root}(B')$, adding a duplicate node v' such that $\operatorname{val}(v) = \operatorname{val}(v')$, $\operatorname{var}(v) = \operatorname{var}(v')$ and if v is a nonterminal then also $\operatorname{low}(v') = \operatorname{low}(v)$ and $\operatorname{high}(v') = \operatorname{high}(v)$, and changing one of the incoming edges of v to point to v' instead.

We have $B \Rightarrow_1 B'$ implies $\operatorname{size}(B) > \operatorname{size}(B')$, and likewise for $\Rightarrow_2$. Hence also for their union we have $B \Rightarrow B'$ implies $\operatorname{size}(B) > \operatorname{size}(B')$. Thus the $\Rightarrow$ relation is well-founded.

A given BDD B has no $\Rightarrow$-successors (which means that no B' exists such that $B \Rightarrow B'$) if and only if B has no redundant nodes and no two distinct similar sub-BDDs, which is the same as saying that B is reduced.

We also have that both redundancy elimination $\Rightarrow_1$ and duplication elimination ($\Rightarrow_2$) preserve denotation, so $B \Rightarrow B'$ implies $D(B) = D(B')$.

Thus, for any BDD $B \in BDD_n$, there exists a reduced BDD $B^\star$ such that $B \Rightarrow^\star B^\star$, where $\Rightarrow^\star$ is the reflexive transitive closure of $\Rightarrow$.

Hence $D(B) = D(B^\star)$ and $\operatorname{size}(B) \geq \operatorname{size}(B^\star)$, with $\operatorname{size}(B) = \operatorname{size}(B^\star)$ if and only if $B = B^\star$.

Now, let $R \in BDD_n$ be an arbitrary BDD.

- Suppose R is reduced. Let $B \in BDD_n$ be a BDD such that $D(R) = D(B)$. Thus there exists a reduced BDD $B^\star$ such that $D(B^\star) = D(B)$ and $D(B^\star) \leq D(B)$. Hence also $D(B^\star) = D(R)$, and by canonicity we get that $R \sim B^\star$. By lemma 30 we get $\operatorname{size}(R) = \operatorname{size}(B^\star)$, and thus $\operatorname{size}(R) \leq \operatorname{size}(B)$, as needed to show that R is minimal.

- Suppose R is minimal. Assume that R is not reduced, then there exists a reduced BDD $R^\star$ with $D(R) = D(R^\star)$ and $\text{size}(R^\star) < \text{size}(R)$. Contradiction to R being minimal. □

Note that, strictly speaking, our definition of *reduced* BDDs differs from the standard definition(s) in the BDD literature. In [Bry86], Bryant defined reduced BDDs using a different equivalence relation than the similarity relation we employ in definition 24, namely an *isomorphism* relation. We paraphrase the definition of this isomorphism relation in definition 32:

Definition 32 (BDD Isomorphism)**.** *BDDs B and B' are isomorphic if there exists a one-to-one function σ from the nodes of B onto the nodes of B' such that for any node v if $\sigma(v) = v'$, then either both v and v' are terminal nodes with $\text{val}(v) = \text{val}(v')$, or both v and v' are nonterminal nodes with $\text{var}(v) = \text{var}(v')$, $\sigma(\text{low}(v) = \text{low}(v'))$ and $\sigma(\text{high}(v) = \text{high}(v'))$.*

Note that BDD isomorphism is an equivalence relation, and moreover note that it is a subset of our BDD similarity relation—if two BDDs are isomorphic then they are similar, but in general not all similar BDDs are isomorphic.

In [Bry86], reduced BDDs are defined as we define them in definition 24, except with the similarity relation replaced by the more fine-grained isomorphism relation. A priori it might seem like our definition yields a different BDD property, but in fact the two definitions are equivalent: clearly the isomorphism-based definition of being reduced implies our similarity-based definition, and in [Bry86] Bryant shows that BDD minimality (definition 28) implies the isomorphism-based definition, so by theorem 31 we get that our similarity-based definition 24 also implies the isomorphism-based definition. Hence the two are equivalent. Therefore we are justified in using definition 24 in place of Bryant's isomorphism-based definition. We choose our similarity-based definition because we find it simpler to formalize, and because the decision trees that this definition is based on are very convenient for proving statements by induction, as we do in lemma 22, for example.

Furthermore, in later BDD literature (e.g. [Knu09] [And97; DS01; Knu09]), including Bryant's own more recent account in [Bry18], yet another equivalent definition is used for reduced BDDs, which does not appeal to an equivalence relation between (sub-)BDDs. Instead, this definition requires there to be at most one terminal node for each of the two Boolean values, and that there be no two distinct nonterminals v, u with $\text{var}(v) = \text{var}(u)$, $\text{low}(v) = \text{low}(u)$ and $\text{high}(v) = \text{high}(u)$. In addition, it requires that no redundant nodes are reachable from root, like we do in definition 24.

This definition has the nice property that all of its conditions are "local" to individual nodes, rather than whole sub-BDDs. However, we still need an equivalence relation on BDDs to state the canonicity property of ROBDDs, which our similarity-based definition 24 already provides "for free".

In chapter 2, we discuss the implementation of algorithms for constructing BDDs with a desired denotation, and for reducing BDDs to their reduced, canonical form.

BDDs allows us to do more with Boolean functions than merely constructing them and comparing them for equivalence—we also have efficient algorithms for finding inputs for which the function yields a given output (`true`/`false`), counting how many such inputs exist, and checking whether the function depends on a given input variable. The fact that BDDs admit efficient algorithms for these tasks demonstrate how closely their concrete structure represents their semantics as Boolean functions.

Of course, BDDs also have their limitations—some Boolean functions can only be represented with BDDs whose size is exponential in the number of variables. [Hun97] gives an interesting example of such a function: the characteristic function for the set of permutations on the set $[n] = \{0, \dots, n-1\}$

is a Boolean function of $n \cdot \log_2 n$ Boolean variables, and the size of a BDD denoting this function is lower-bounded by $2^{n/2}$.

1.3 Calculus of Inductive Constructions

We implement our verified BDD library in Lean 4, a functional programming language and proof assistant, which is based on the Calculus of Inductive Constructions [MU21].

The Calculus of Inductive Constructions is an extension of Dependent Type Theory, which is in turn and expressive and versatile formalism able to represent both computation (executable programs) and mathematical statements along with their proofs [Pau15].

In this section, we look at some Lean code, and describe some of the basic definitions and theorems in Lean that we use in our BDD library.

In Lean, Booleans are defined as a simple inductive type:

```
inductive Bool : Type where
  | false : Bool
  | true : Bool
```

This code defines a type (a term of type `Type`) called `Bool`, with two *constructors*, which are ways of creating terms of the defined type. Namely, there are just two ways: either using `false` or `true`, with no additional information.

This definition is only trivially inductive, since none of the constructors actually require terms of the inductively defined type. The type of natural numbers, `Nat`, does rely on induction, which does more justice with the `inductive` keyword in the definition:

```
inductive Nat where
  | zero : Nat
  | succ : Nat → Nat
```

The next definition demonstrates the use of *dependent* pairs to define the type of natural numbers below some number n, which corresponds to the set $[n]$ in our notation from section 1.1:

```
def Fin (n : Nat) := { i // i < n }
```

Read: given a term `n` of type `Nat`, `Fin n` is the type of (dependent) pairs `⟨i, h⟩` where `i` is a `Nat` and `h` is a term of type `i < n`, which we interpret as a *proof* of the *proposition* `i < n`.

Note how Lean allows us to specify just enough type annotations for it to infer the types of terms in the definition, and leave the rest of the information implicit.

In general, for a pair `p = ⟨l, r⟩`, we use the notation `p.1` to access the first constituent `l`, and `p.2` to access the second constituent `r`.

Another type that we use throughout our BDD library in `Vector`, which is the type of *arrays of a given length*. Lean provides a basic `Array` type that represents contiguous memory arrays that are homogeneous in the sense that all elements of the array are of one given type. The type `Vector α n` represents arrays of length `n` consisting of terms of type α.

In particular, we represent n-ary functions as functions that take a `Vector` of length n:

```
abbrev Func n α β := Vector α n → β
```

This defines `Func n α β` as an abbreviation for the type of functions from n input vectors of αs to an output β.

In particular, we can define the type of n-ary Boolean functions as:

```
abbrev BoolFunc n := Func n Bool Bool
```

This is not the only possible way to represent Boolean function in Lean. First, Lean defines a type `BitVec` of "bit vectors", which is equivalent to `Vector Bool`. The `BitVec` type has a specialized implementation using an underlying `Nat`, which makes it potentially more compact and efficient than `Vector Bool`. While we can represent n-ary Boolean functions as `BitVec n → Bool` instead of our definition, which amounts to `Vector Bool n → Bool`, we choose `Vector Bool` because it allows us to define general notions about n-ary functions, such as *restrictions* (definition 3) and *dependencies* (definition 4) at the right level of generality, whereas `BitVec` confines us to definitions about Boolean functions in particular.

Also, the advantages of using `BitVec`s in terms of performance are only relevant for programs that make heavy use of Boolean/bit vectors. While the proofs in our library include plenty of *reasoning* about such vectors, for most BDD use cases these vector never occur at runtime, and so their representation has no effect on performance whatsoever. For example, checking if two BDDs denote the same Boolean function does not involve any actual input vectors. (See section 2.1.2 for more details.)

Another alternative representation for n-ary functions would be to use the `Function.OfArity` type from Lean's mathematics library, *Mathlib*. This type uses functions from `Fin n` to α to represent the `n` inputs, each of type α, of an n-ary function. Such `Fin n` → α functions are also known as *tuples* in Lean. For the purposes of our library, we found vectors to be more convenient than tuples due to the vast number of operations and existing lemmas that Mathlib provides for working with vectors.

Next, we formalize the notion of dependency on an input variable from definition 4:

```
def IndependentOf (f : Func n α β) (i : Fin n) := ∀ a v, f v = f (Vector.set v i a)

def DependsOn (f : Func n α β) (i : Fin n) := ¬ IndependentOf f i
```

Thus a term of type `IndependentOf f i` is a proof that for all input vectors `v` and terms `a` of the input type α, the function `f` returns the same output for `v` as it does for `Vector.set v i a`, which is `v` with the `i`th index set to `a`. A term of type `DependentOf f i` says exactly the opposite—which is the same as saying that there exists some input vector and two different values such that the output of the function changes if one value appears instead of the other.

We define the set of input indices that a given n-ary function depends on, as a subtype of `Fin n`:

```
def Dependency (f : Func n α β) := { i // DependsOn f i }
```

We also formalize the notion of a restriction from definition 3:

```
def restrict (f : Func n α β) : α → Fin n → Func n α β := fun a i I ↦ f (I.set i a)
```

And we formally prove in Lean that the restriction of a function at a given index is independent of that index:

```
lemma restrict_independentOf : IndependentOf (restrict f c i) i := by simp
```

Here, the proof term after the `:=` is a *tactic* proof. We use the powerful *simplification* tactic, `simp`, which is able to produce a concrete proof term all by itself in this case.

In contrast, the proof of the following lemma, which we use later in our proofs about BDDs is far more involved:

```
lemma eq_of_forall_dependency_getElem_eq {f : Func n α β} {I J : Vector α n} :
    (∀ (x : Dependency f), I[x] = J[x]) → f I = f J := by
  induction n with
  | zero =>
    intro h
    congr
    ext i hi
    contradiction
  | succ n ih =>
    intro h
    let g : Vector α n → β := fun v ↦ f (Vector.push v I[n])
    have h2 : ∀ V : Vector α (n + 1), I[n] = V[n] → f V = g V.pop := by
      ...
    by_cases hf : DependsOn f n
    ...
```

The lemma states that `f I = f J` whenever `I` and `J` agree on all indices that `f` depends on. The proof proceeds by induction on n. The base case of $n = 0$ is simple—we use the `congr` tactic to reduce the goal from proving `f I = f J` to proving `I = J`, then we compare `I` and `J` index by index by introducing an arbitrary index `i`, of which there are none since the vectors are both of length 0, and we get a `contradiction` that proves the goal.

In the induction step, we get an inductive hypothesis `ih` that says that the lemma holds for n-ary functions, and we use it to prove the lemma for $n + 1$-ary functions. We define a specific n-ary helper `g`—to which we apply the inductive hypothesis—in terms of our original $n + 1$-ary `f`, and we proceed by case distinction on whether or not `f` depends on its last input index by invoking the `by_cases` tactic. Note that this tactic appeals to the classical principle of excluded middle, which is needed since we are proving this lemma for arbitrary types α and β. However, for finite input types and for output types with decidable equality, which we can readily express in Lean as well, we can *decide* whether a function depends on a given input index without requiring the law of excluded middle, simply by checking all possible inputs and comparing the corresponding outputs. But this naive procedure is not very practical—already for Boolean functions, this means checking 2^n inputs for an n-ary function. Fortunately, as we show in section 2.1.4, using BDDs we obtain a much more efficient algorithm for deciding function dependencies.

For expressing that a proposition is decidable, Lean provides the `Decidable` type class:

```
class inductive Decidable (p : Prop) where
  | isFalse (h : ¬ p) : Decidable p
  | isTrue  (h :   p) : Decidable p
```

This definition says that there are two ways to show that a proposition `p` is decidable: either by claiming that it holds (applying the `isTrue` constructor) and providing a proof that it indeed holds, or by claiming that it does not hold and showing that.

Hence if we have a type α and a predicate `P` of type α → `Prop`, we interpret a function `f` of type `(a : `α`) → Decidable (P a)` as a *decision procedure* for the predicate `P`. Lean also provides the abbreviation `DecidablePred P` which expands to `(a : `α`) → Decidable (P a)`, saying that the predicate `P` is decidable, and similarly `DecidableRel R` says that the relation `R` is decidable.

The fact that `Decidable` is type *class* just means that we can register `Decidable`-valued functions (decision procedures) by defining them with the `instance` keyword, and then Lean tries to use these "instances" to implicitly synthesize a `Decidable` term when it needs one. In particular, Lean's `decide` proof tactic attempts to prove a proposition `p` by synthesizing a `Decidable p` term and checking that it is an `isTrue`.

In section 2.1, we show the `Decidable` instances that our library provides, which allow users to prove certain statements about Boolean functions by simply invoking the `decide` tactic.

Chapter 2

Binary Decision Diagrams in Lean

This chapter describes our Lean 4 BDD library. We begin in section 2.1 with the library's user-facing interface and the facilities it provides for reasoning about Boolean functions. In section 2.2, we investigate the underlying BDD formalization in Lean, as well as the implementation and verification of key BDD algorithms. In section 2.3, we present a BDD-based SAT solver as an example application of our library.

The source code for our library is available at `https://github.com/eshelyaron/lean4-bdd`. The library is structured as follows:

- `BDD.lean` provides the main interface of the library, which we discuss in section 2.1.
- `Nary.lean` includes definitions and general results about n-ary functions, some of which we have described in section 1.3.
- `DecisionTree.lean` includes definitions and results about decision trees (see definition 18).
- `Basic.lean` includes our basic BDD construction and various results about BDDs. We examine key parts of this file in sections 2.2.1 and 2.2.2.
- `Apply.lean` implements a variant of the BDD *Apply* algorithm from [Bry86], which synthesizes a composed BDD from two existing BDDs and a binary Boolean function. See section 2.2.5 for more details.
- `Choice.lean` implements and proves the correctness of an algorithm for finding inputs for which the denotation of a given BDD returns `true`. It is described in section 2.2.7.
- `Collect.lean` implements and proves the correctness of a DFS algorithm that collects all nonterminal nodes reachable from the root of a given OBDD.
- `Lift.lean` implements and proves the correctness of a simple BDD type-lifting operation, which produces a BDD $\mathsf{lift}(B) \in BDD_{n'}$ from a BDD $B \in BDD_n$ under the assumption that $n \leq n'$. See section 2.2.3 for more details.
- `Reduce.lean` implements the BDD *Reduce* algorithm from [Bry86], which transforms an arbitrary OBDD to an ROBDD with the same denotation.

- `Relabel.lean` implements and proves the correctness of a BDD relabeling operation, which changes the variable indices of a given OBDD according to a given relabeling function.
- `Restrict.lean` implements an algorithm for producing a BDD representing the restriction $f_{i \leftarrow b}$ of a Boolean function f given a BDD for f itself.
- `Sim.lean` implements a decision procedure for BDD similarity.
- `Size.lean` includes definitions and results about the size of OBDDs, per definition 10.
- `Sat.lean` is an example application of the library—it implements a SAT solver via translation of propositional formulae in CNF to BDD, and proves the correctness of the translation to obtain proofs of (un)satisfiability.
- `Trim.lean` implements and proves the correctness of a “garbage collection” operation, which eliminates unreachable nodes from a given BDD.

With the exception of two lemmas in `Reduce.lean`, all of the Lean definitions and statements that we present in this chapter are fully formalized and proved correct. We explicitly note the lemmas whose formal proofs are not yet complete when we discuss the BDD reduction algorithm in section 2.2.5.

Figure 2.1 shows the dependencies between the different files in our library. The red circle around `Reduce` expresses that, in contrast with all other files, not all formal proofs in `Reduce.lean` are complete.

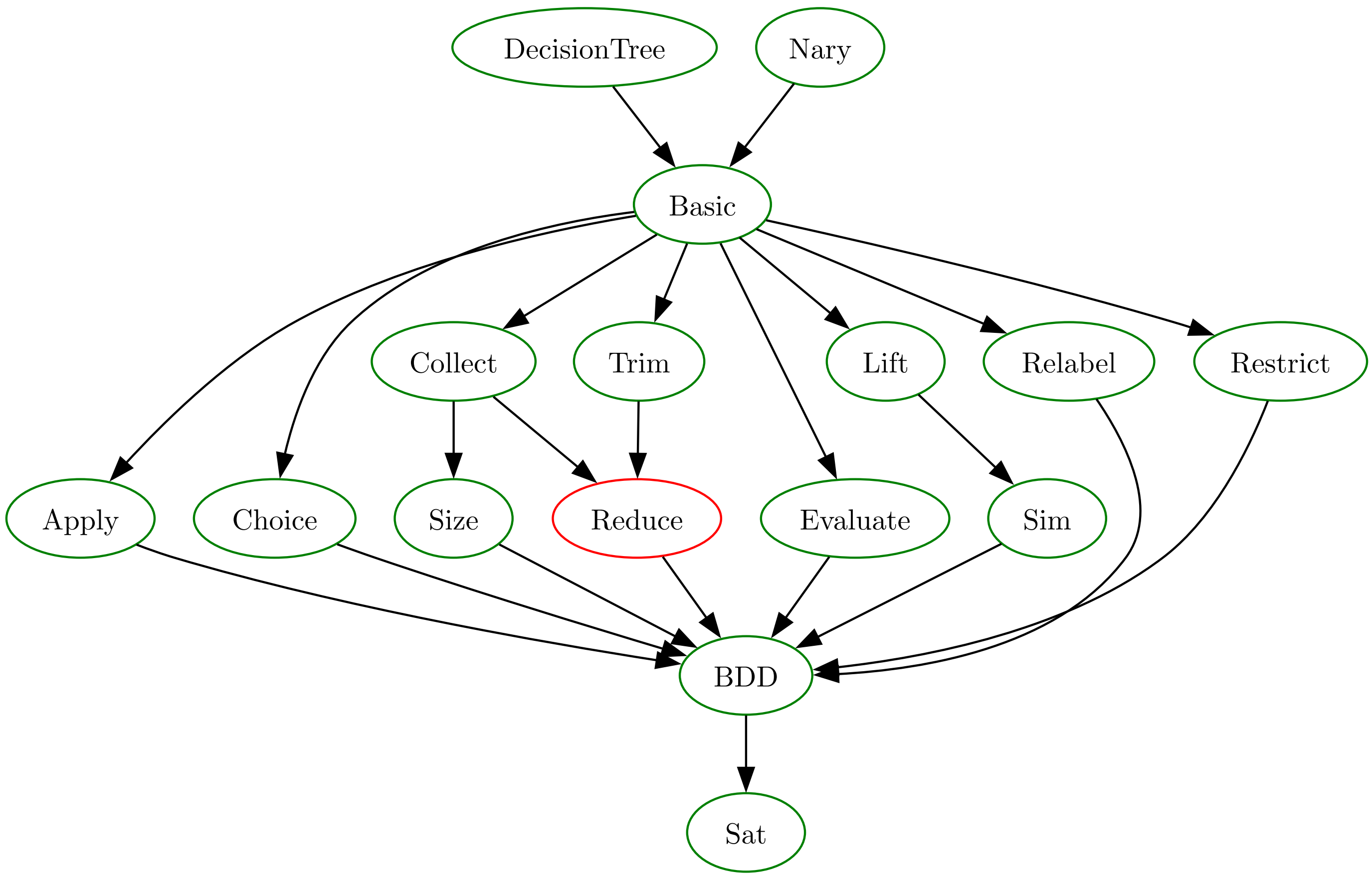

Figure 2.1: File-level dependencies in our BDD library

2.1 Interface

The interface of the library is a subset of the "Boolean Function API" described in [Bry18]. It is geared towards facilitating operations on Boolean functions that are often used in model checking applications, such as *efficiently deciding semantic equivalence* of logical formulae.

2.1.1 The Abstract BDD Type

Our library's interface revolves around a type `BDD`, which represents a reduced and ordered BDD.

```
structure BDD where
  nvars : Nat
  ...
```

Note that, unlike definition 8 wherein the set of BDDs BDD_n is indexed by a natural number n—the number of input variables in the denotation of elements of BDD_n—the `BDD` type in Lean represents (RO)BDDs of all variable numbers. Namely, we can think of `BDD` as corresponding to the union $\bigcup_{n \in \mathbb{N}} BDD_n$. Instead of indexing the type of BDDs by their input sizes, the `BDD` structure has a public field `nvars`, which provides the input size of given `BDD` as a piece of data (available at runtime).

The use of a single type for BDDs of different input sizes contributes to the flexibility of the interface, by allowing users to build up BDDs without committing to a number of input variables ahead of time, while having the library keep track of this detail for them.

Besides the public field `nvars`, `BDD`s have additional *private* fields which are internal implementation details, not exposed as part of the library's API. (We describe these implementation details in section 2.2.)

Given a BDD of n variables $B \in BDD_n$ and a natural number n' such that $n \leq n'$, we can *lift* B to a BDD of n'-variables $lift_{n'}(B) \in BDD_{n'}$, simply by considering each variable index $i \in [n]$ that occurs in B as an index in $[n']$ instead of $[n]$. The resulting $\mathsf{lift}_{n'}(B)$ is "faithful" to B in the sense that evaluating $\mathsf{lift}_{n'}(B)$ with respect to inputs $x_0, \ldots, x_n, \ldots, x_{n'}$ yields $D(B)(x_0, \ldots, x_n)$ where D is the BDD denotation function from definition 11. Therefore, we can naturally evaluate B with respect to n' inputs by lifting B and then and evaluating. This observation allows us to formalize the notion of semantic equivalence between BDDs of potentially different number input sizes, which is very useful in practice since it allows users to compare any two BDDs that they construct, without worrying about the finer details of whether or not the input sizes match. We define the *denotation* of `BDD`s accordingly:

Each `BDD B` *denotes* a Boolean function which maps (at least) `B.vnars` input Booleans to one output Boolean. Formally, we define the denotation of `B` as a family of Boolean functions, one for each input size greater or equal to `B.nvars`.

```
def denotation (B : BDD) (h : B.nvars ≤ n) : BoolFunc n := (B.lift h).evaluate
```

We also provide an abbreviated name for the common case of using `denotation` with `n = B.nvars`:[1]

```
abbrev denotation' O := denotation O (le_refl _)
```

We discuss the implementations of the `lift` and `evaluate` functions later in section 2.2.3, but we can already see how the definition of `B.denotation h` corresponds to $D(\mathsf{lift}_n(B)) = x_0, \ldots, x_n \mapsto D(B)(x_0, \ldots, x_{\mathsf{B.nvars}})$. If `n = B.nvars`, then $\mathsf{lift}_n(B)$ is just B and so we get back to $D(B)$. This is expressed by the following lemma in Lean:

[1]All Lean definition in this section are in the `BDD` namespace, so `denotation` is accessed as `BDD.denotation`, etc.

```
lemma lift_refl {B : BDD} : (B.lift (le_refl _)) = B := by simp [lift]
```

For a BDD $B \in BDD_n$, $D(\mathsf{lift}_{n'}(B))$ is independent of the last $n' - n$ inputs—it may only depend on the first n variables. This is formalized in the following lemma:

```
lemma denotation_independentOf_of_geq_nvars {B : BDD} {i : Fin n} {h : B.nvars ≤ n} :
  B.nvars ≤ i → IndependentOf (B.denotation h) i
```

We can read the lemma `denotation_independentOf_of_geq_nvars` as saying that for any `BDD B`, the value of the denotation of `B` given some input vector `I` is the same regardless of the value of `I` at any index `i` greater or equal to `B.nvars`. Hence `B.nvars` is an upper bound on the number of variables the denotation of `B` depends on.

From the perspective of an external consumer of the library's API, that is *almost* all there is to know about terms of type `BDD` as such. Namely, a `BDD B` denotes a family of Boolean functions, which may only depend on the first `B.nvars` input variables.

2.1.2 Semantic Equivalence

In section 2.1.3 we present the library's facilities for constructing `BDDs`, but first we show how to efficiently check two `BDD` that we already have at hand for semantic equivalence, which is one of the main use cases for this library.

We formally define a semantic equivalence relation on `BDDs`, which holds for `BDDs B` and `C` iff they denote the same function:

```
def SemanticEquiv : BDD → BDD → Prop := fun B C ↦
  B.denotation (Nat.le_max_left  ..) = C.denotation (Nat.le_max_right ..)
```

Note that, in this definition, we implicitly require the two `BDDs` to denote the same function for inputs of size $\max\{\mathsf{B.nvars}, \mathsf{C.nvars}\}$, which in turn implies that they denote the same function for larger inputs as well. We formally prove a slightly stronger version of this implication in the following lemma, which says that if we know that the denotations of two BDDs coincide for some input size, then they coincide for any other input size as well:

```
lemma denotation_eq_of_denotation_eq {B C : BDD}
    (hn : B.nvars ⊔ C.nvars ≤ n) (hm : B.nvars ⊔ C.nvars ≤ m) :
  B.denotation (n := n) (by omega) = C.denotation (n := n) (by omega) →
  B.denotation (n := m) (by omega) = C.denotation (n := m) (by omega)
```

We use `denotation_eq_of_denotation_eq` to show that `SemanticEquiv` is indeed an equivalence relation, which requires amounts to proving reflexivity, symmetry and transitivity:

```
theorem SemanticEquiv.equivalence : Equivalence SemanticEquiv :=
  { refl := fun _ ↦ rfl,
    symm := fun h ↦ Eq.symm (denotation_eq_of_denotation_eq (by omega) (by omega) h),
    trans := by
      intro B C D hBC hCD
      simp_all only [SemanticEquiv]
      let m := max (max B.nvars C.nvars) D.nvars
      apply denotation_eq_of_denotation_eq (n := m) (by omega) (by omega)
      trans C.denotation (by omega)
      · exact denotation_eq_of_denotation_eq .refl (by omega) hBC
      · exact denotation_eq_of_denotation_eq .refl (by omega) hCD
  }
```

Thanks to the underlying implementation of the BDD type as actual BDDs, we can efficiently decide the SemanticEquiv relation by exploiting the *canonicity* of reduced ordered BDDs. From the library's API perspective, this is manifested as a DecidableRel instance:

```
instance instDecidableSemanticEquiv : DecidableRel SemanticEquiv
  ...
```

We thus obtain the following general "recipe" for deciding semantic equivalence of propositional formulae (or other structures that denotes Boolean functions) in Lean, using the provided BDD interface:

1. Define your type of formulae Φ,
2. define a semantic equivalence relation $\cdot \equiv \cdot$ on Φ,
3. define a translation function f from Φ to BDD using the BDD construction API described in section 2.1.3,
4. prove a correctness lemma C saying that your translation function f respects semantic equivalence, in the sense that $p \equiv q$ if and only if $f(p)$ and $f(q)$ are semantically equivalent BDDs
5. now, given two formulae p, q, we can obtain a proof of their (in)equivalence by applying `instDecidableSemanticEquiv (f p) (f q)`, and transporting the result back via C to obtain a proof of $p \equiv q$ (or the negation thereof).

We return to instDecidableSemanticEquiv and discuss its implementation in section 2.2.4.

2.1.3 Constructing BDDs

We construct and reason about BDDs via interface functions with a clear characterization in terms of the denotation of the constructed BDD.

The basic building blocks are two kinds of BDDs with simple denotations:

1. BDDs that denote the constant Boolean functions $\mathbf{0}$ and $\mathbf{1}$, and
2. BDDs that denote the projection functions $\boldsymbol{x}_1$ for of a single decision variable i.

We provide the interface functions const and var which implement these basic building blocks:

```
def const : Bool  → BDD
def var   : Nat   → BDD
```

The following lemma characterizes const in terms of the denotation of the BDD it produces:

```
lemma const_denotation : (const b).denotation h = Function.const _ b
```

This lemma, which is part of the library's API, establishes the use of const to obtain BDDs that denote a constant Boolean function.

Similarly, we provide the expected specification lemma for the denotation of BDDs produced via var:

```
lemma var_denotation : (var i).denotation h I = I[i]
```

In addition to the basic building blocks, `const` and `var`, we provide composition functions for combining `BDDs` to construct `BDDs` that denote more complex Boolean functions. The composition functions correspond to the logical connectives $\wedge, \vee, \otimes, \rightarrow$ and $\neg$:

```
def and : BDD → BDD → BDD
def or  : BDD → BDD → BDD
def xor : BDD → BDD → BDD
def imp : BDD → BDD → BDD
def not : BDD → BDD
```

The simplicity of the types of these functions is where our decision to have a single `BDD` type for BDDs of all input sizes really pays off, since users can freely compose BDDs of different input sizes. For example, the following shows the construction of a BDD denoting the ternary majority function (whose input size is 3, see section 1.1), by composing BDDs of smaller input size:

```
def majority3 :=
  (or (or (and (var 0) (var 1)) (and (var 0) (var 2))) (and (var 1) (var 2)))

example : majority3.nvars              = 3 := rfl
example : (and (var 0) (var 1)).nvars = 2 := rfl
example : (var 0).nvars                = 1 := rfl
```

Furthermore, we can decide whether the `BDDs` that we construct are semantically equivalent, without ever mentioning their input sizes. In the following examples, we pass the `+native` argument to the `decide` tactic, which instructs Lean to use its native code complier to run the decision procedure `instDecidableSemanticEquiv`. This is required because Lean's *elaborator*, which is the default mechanism that the `decide` uses for evaluation, is currently unable to execute some the functions in our implementation of `instDecidableSemanticEquiv` (which we describe in section 2.1.2).

```
-- The majority function is not constantly true.
example : ¬ majority3.SemanticEquiv (const true) := by decide +native

-- It is not constantly false, either.
example : ¬ majority3.SemanticEquiv (const false) := by decide +native

-- Another BDD, with a suggestive name.
def majority3' :=
  (and (imp (var 0) (or (var 1) (var 2))) (imp (var 0).not (and (var 1) (var 2))))

-- The two ways of writing the majority function are equivalent.
example : majority3.SemanticEquiv majority3' := by decide +native
```

Of course, some use cases do require reasoning about the input size of BDDs. For example, in order to evaluate a BDD with respect to a given input vector, we must be able to show that the size of the vector is greater or equal to the input size of the BDD, since otherwise the BDD evaluation may depend on indices that are out of the bounds of the input vector.

We therefore provide, as part of the library's API, lemmas that characterize the input sizes of the BDDs that the interface functions produce, starting with the building blocks `const` and `var`:

```
lemma const_nvars : (const b).nvars = 0
lemma var_nvars : (var i).nvars = i + 1
```

Each composition function is provided along with a lemma that expresses the `nvars` of the output `BDD` as a function of the `vnars` of the input `BDD`(s). Namely, for the binary functions, the output `nvars` is naturally the maximum of the `nvars` of the inputs, while the unary function `not` preserves `nvars`:

```
lemma and_nvars {B C : BDD} : (B.and C).nvars = B.nvars ⊔ C.nvars
lemma or_nvars  {B C : BDD} : (B.or  C).nvars = B.nvars ⊔ C.nvars
...
lemma not_nvars {B : BDD} : B.not.nvars = B.nvars
```

In addition, the composition functions come with associated specification lemmas, which characterize the denotation of the output `BDD` in terms of the denotations of the inputs:

```
lemma and_denotation {B C : BDD} {I : Vector Bool n} {h} :
  (B.and C).denotation h I =
  ((B.denotation (by simp_all) I) && (C.denotation (by simp_all) I))

lemma or_denotation {B C : BDD} {I : Vector Bool n} {h} :
  (B.or C).denotation h I =
  ((B.denotation (by simp_all) I) || (C.denotation (by simp_all) I))

...

lemma not_denotation {B : BDD} {I : Vector Bool n} {h} :
  B.not.denotation h I = ! B.denotation (by simp_all) I
```

These lemmas are to be used by consumers of the API to prove that a mapping from some type of interest Φ to `BDD`s respects semantic equivalence, which in turns facilitates the use of `instDecidableSemanticEquiv` to efficiently decide semantic equivalence in Φ.

All of these Boolean operators are implemented as specializations of a single general function called `apply`, which takes a binary Boolean function `op` and two BDDs `B` and `C`, and returns a BDD whose denotation returns `op` applied to the output of `B` and `C`:

```
def apply : (Bool → Bool → Bool) → BDD → BDD → BDD

lemma apply_nvars {B C : BDD} {op} : (apply op B C).nvars = B.nvars ⊔ C.nvars

lemma apply_denotation {B C : BDD} {op} {I : Vector Bool n} {h} :
  (apply op B C).denotation h I =
  (op (B.denotation (by simp_all) I) (C.denotation (by simp_all) I))
```

In addition to `apply` and the functions derived from it, we also provide a function that computes the restriction $f_{i \leftarrow b}$ of a Boolean function f at a given index i to a given Boolean value b:

```
def restrict (b : Bool) (i : Nat) (B : BDD) : BDD
```

The arguments that `restrict` takes are a Boolean `b`, a `BDD` `B` and an index `i`. According to our definition 3, the index `i` should be $i \in [n]$ where n is the arity of the restricted function. This suggests defining `i` as a `Fin B.nvars`, while in the definition above we see `i` defined as a plain `Nat`. We do so for flexibility—by keeping the type of the index independent of the `BDD` we are restricting, it becomes easier to, for example, restrict a `BDD` repeatedly with a list of indices of type `List Nat`. If we were to define `i` as a `Fin B.nvars`, then to restrict again with another index `i'` we would need `i'` to be in `Fin (B.restrict b i).nvars`, which is equivalent to the `Fin B.nvars` we started with (by lemma `restrict_nvars` below), but still not the same type.

For indices `i < B.nvars`, the `restrict` function exactly corresponds to definition 3. We generalize to larger indices, on which the restricted function cannot actually depend, by simply defining `B.restrict b i = B` when `i ≥ B.nvars`:

```
lemma restrict_geq_eq_self {B : BDD} : i ≥ B.nvars → B.restrict b i = B
```

We also provide interface lemmas that specify the behavior of `restrict` in terms of its affect on the input size and the denotation of the restricted BDD:

```
lemma restrict_nvars {B : BDD} {i} : (B.restrict b i).nvars = B.nvars

lemma restrict_denotation {B : BDD} {I : Vector Bool n} {i} {hi : i < n} {h} :
  (B.restrict b i).denotation h I =
  (Nary.restrict (B.denotation (restrict_nvars ▸ h)) b ⟨i, hi⟩) I
```

We further use `restrict` along with binary operators to facilitate *quantification* over Boolean variables:

```
def bforall (B : BDD) (i : Nat) : BDD :=
  (and (B.restrict false i) (B.restrict true i))

def bexists (B : BDD) (i : Nat) : BDD :=
  (or  (B.restrict false i) (B.restrict true i))
```

These functions are also accompanied by corresponding interface lemmas:

```
lemma bforall_nvars {B : BDD} {i} : (B.bforall i).nvars = B.nvars

lemma bforall_denotation {B : BDD} {i} {hi : i < n} {I : Vector Bool n} {h} :
  (B.bforall i).denotation h I =
  (∀ b, B.denotation (by simp_all) (I.set i b) : Bool) := by simp_all [bforall]
```

In `bforall_denotation` we leverage Lean's conversion between `Prop` and `Bool`: Lean knows that for `Bool`s, the `Prop` `∀ b, P b` is the same as `P true ∧ P false`, which in our case is a conjunction between two `Bool`s, which is again just a `Bool`, so we can readily compare the two sides of the equation for equality as `Bool`s.

In model checking applications, we often want to quantify over multiple input variables at once [CG18]. To facilitate that use case, we provide the following generalized functions:

```
def bforalls (B : BDD) (l : List Nat) := List.foldl bforall B l

def bexistss (B : BDD) (l : List Nat) := List.foldl bexists B l
```

Our implementation of `bforall`, `bexists` and the above generalized variants in terms of `and`, `or` and `restrict` is clear and concise, but it is not the most efficient. More efficient algorithms exist for quantifying over one or more Boolean variables with BDDs, without constructing BDDs and combining BDDs for restrictions first [Bry18]. Since such quantification is essential in many model checking applications (see [Bur+94; Yan+98; Gat18], for example), we hope to improve our implementation in these directions in the future (see also section 3.1).

Another crucial ingredient for BDD-based model checking is the ability to *relabel* variables, which is a necessary step in BDD-based computation of the image (and as well as preimage) of a relation [CG18].

Relabeling means changing the input indices that a function depends on in a consistent manner. We can imagine that each input has its index attached to it as a label, and our function uses those labels to determine which input is which. When we relabel, we give some inputs new labels, keeping track of the correspondence between new labels and old ones. It does not make sense to give two inputs the same label, but we can detach an old label from one input and attach it to another as its new label. We then equip our function with a dictionary that tells it the new label where it can find the input corresponding to each old label it depends on.

```
def relabel (B : BDD)
    (f : Fin B.nvars → Fin n)
    (h : ∀ i i' : (Dependency B.denotation'), i.1 < i'.1 → f i.1 < f i'.1) :
  BDD
```

The `relabel` function takes a `BDD` and a relabeling function `f`, as well as a precondition `h`, and it produces a relabeled BDD, in which the `i`th input is receives the new label `f i`.

The precondition `h` asserts that `f` is strictly increasing for input indices that the denotation of `B` depend on. Obviously, `f` must be injective for such indices, since otherwise we would have two relevant inputs with the same new label (index). Injectivity is implied by the requirement that `f` is strictly increasing. We need `f` to be strictly increasing rather than merely injective, to ensure that the resulting BDD remains *ordered*.

As with other `BDD`-valued functions, we provide lemmas that characterize the input size and denotation of `relabel B f h` in terms of those of `B`:

```
lemma relabel_nvars {B : BDD} {f : _ → Fin n} {h} : (relabel B f h).nvars = n

lemma relabel_denotation {B : BDD} {f} {hf} {I : Vector Bool n} {h} :
  (relabel B f hf).denotation h I = B.denotation' (Vector.ofFn (fun i ↦ I[f i]))
```

The lemma `relabel_denotation` can be read as saying that evaluating the relabeled BDD with respect to an input vector `I` is the same as evaluating the original BDD with respect to the vector in which the value under label (in index) `i` is the value under the corresponding new label `f i` in `I`.

As an example for the use of `relabel`, we can define the functions `mv` and `cp` from [Gat18], which take a BDD of some variables V to a "double domain" $V \cup V'$ with two copies v, v' for each $v \in V$, as follows:

```
def mv (B : BDD) :=
  relabel B (n := 2 * B.nvars + 1) (fun i ↦ ⟨2 * i.1,     by omega⟩) (by simp)

def cp (B : BDD) :=
  relabel B (n := 2 * B.nvars + 1) (fun i ↦ ⟨2 * i.1 + 1, by omega⟩) (by simp)
```

We can also define the inverse functions `unmv` and `uncp`, which send a BDD B over $V \cup V'$ to a BDD over V. `unmv` assumes that B only depends on variables in V (even indices), while `uncp` assumes that B only depends on variables in V' (odd indices):

```
def unmv (B : BDD) : (∀ i : Dependency B.denotation', Even i.1.1) → BDD := fun h ↦
```

```
  relabel B (n := (B.nvars + 1) / 2) (fun i ↦ ⟨i / 2,       by omega⟩) (by ...)

def uncp (B : BDD) : (∀ i : Dependency B.denotation', Odd  i.1.1) → BDD := fun h ↦
  relabel B (n := (B.nvars + 1) / 2) (fun i ↦ ⟨(i - 1) / 2, by omega⟩) (by ...)
```

2.1.4 Variable Dependencies

As mentioned in section 1.3, we can decide whether an arbitrary n-ary Boolean function f depends on a given input index i by going over all 2^n possible inputs x and checking if changing the value of x_i changes the value of $f(x)$. In contrast, if f is represented by a ROBDD B with $D(B) = f$, then can check whether f depends on i more efficiently, by scanning the size(B) nodes in B for a node v with $\text{var}(v) = i$. We implement this operation and prove its correctness as part of our library, and we encapsulate it in the following `DecidablePred` instance:

```
instance instDecidableDependsOn (B : BDD) : DecidablePred (DependsOn B.denotation')
```

For example, we can use `instDecidableDependsOn` to show that the ternary majority function depends on its first input (at index 0), while its restriction at index 0 to any Boolean b does not:

```
example : DependsOn (majority3.denotation') ⟨0, by simp⟩ := by
  decide +native

example : ∀ b, ¬ DependsOn ((majority3.restrict b 0).denotation') ⟨1, by simp⟩ := by
  decide +native
```

2.1.5 Finding Solutions

In section 2.1.3, we saw an example of how `instDecidableSemanticEquiv` lets us decide whether the denotation of a BDD has any satisfying inputs, by checking whether it is semantically equivalent to `const false`. Given a BDD $B \in BDD_n$, we can also efficiently obtain a concrete input vector $x \in \mathbb{B}^n$ such that $D(B)(x) = \text{true}$, if such an x exists.

For an arbitrary n-ary Boolean function f, even if we are guaranteed that f is satisfiable, to find a satisfying input we basically have no choice but to check all 2^n possible inputs until we find one. In contrast, with a BDD representation of f we can find such an input in at most n steps [Knu09], which amounts to an exponential speed up.

We provide two interface functions for obtaining inputs that satisfy the denotation of a given BDD. The first is a `choice` function, which is applicable when we know that the denotation of a given BDD is satisfiable:

```
def choice {B : BDD} (s : ∃ I, B.denotation' I) : Vector Bool B.nvars
```

The following lemma provides the correctness guarantee that the output of `choice` satisfies the denotation of `B`:

```
lemma choice_denotation {B : BDD} {s : ∃ I, B.denotation' I} :
  B.denotation' (B.choice s) = true
```

In addition to `choice`, we provide an `Option`-valued function called `find`, which returns an input that satisfies the denotation of a given BDD if such an input exists, otherwise it returns `none` to signal that there is no such input:

```
def find {B : BDD} : Option (Vector Bool B.nvars)

lemma find_none {B : BDD} : B.find.isNone → B.denotation' = Function.const _ false

lemma find_some {B : BDD} {I} : B.find = some I → B.denotation' I = true
```

For example, we can use `find` to find inputs that satisfy the ternary majority function:

```
#eval! majority3.find.bind (fun I ↦ some I.toList) -- some [false, true, true]
```

This concludes our discussion of the library's interface. The most important principle in our design of the described interface is that it must allow users to construct BDDs and reason (prove theorems) about their denotations, while knowing nothing about the implementation details of BDDs beyond the fact that they denote Boolean functions. The various lemmas that we provide facilitate reasoning purely in terms of Boolean functions, while benefiting from the efficient implementation that the underlying BDDs admit.

2.2 Implementation and Verification

In this section, we look under the hood of the abstract `BDD` type that we presented in section 2.1, and discuss its implementation in Lean and the formal verification of the BDD operations the library provides.

2.2.1 Structures

In addition to the `nvars` public field that we have introduced above, the `BDD` type has three more private fields called `nheap`, `obdd` and `hred`:

```
structure BDD where
  nvars         : Nat
  private nheap : Nat
  private obdd  : OBdd nvars nheap
  private hred  : obdd.Reduced
```

The `obdd` field holds the underlying Ordered BDD structure. Our concrete representation for OBDDs is a family of types `OBdd n m`, indexed by two natural numbers: `n` is the input size of the BDD and `m` is the size of the memory array allocated for the BDD, which we call the "heap" of the BDD—so `m` is the maximum number of nonterminal nodes the BDD may contain. The abstract `BDD` type conceals these type indices behind the `nvars` and `nheap` fields to provide a single uniform type for BDDs of all sizes, but for implementation and formalization purposes it is convenient to have more specific types for BDDs with given input and heap sizes.

The `hred` field of a `BDD` holds a proof that the underlying OBDD is also *reduced* as per definition 24. We come back to our definition of `Reduced` after explaining the `OBdd` construction in more detail.

The `OBdd` type is naturally defined as a subtype of a more general type of possibly non-ordered BDDs, which we call `Bdd`:

```
def OBdd n m := { B : Bdd n m // B.Ordered }
```

Note the difference in capitalization between `BDD` and `Bdd`—the former is our abstract type representing ROBDDs, while the latter is the underlying "raw" BDD type, whose inhabitants need not be ordered nor reduced.

Our Bdd type is a concrete representation of the BDDs we defined in definition 8 in terms of abstract graphs and functions. We first examine the definition of Bdd and then show how it induces the graph structure from definition 8. Similarly to OBdd, Bdd is actually defined as a family of types, indexed by two natural numbers which we think of as the BDD input size (n) and the number of nonterminal nodes in the BDD graph (m). So a Bdd n m represents a BDD $B \in BDD_n$ with m nonterminal nodes. (This implies $\text{size}(B) \leq m$, but not necessarily $\text{size}(B) = m$, since some of the m nonterminal nodes may not be reachable from $\text{root}(B)$.)

```
inductive Pointer m where
  | terminal : Bool  → Pointer _
  | node     : Fin m → Pointer m

structure Node (n) (m) where
  var  : Fin n
  low  : Pointer m
  high : Pointer m

structure Bdd (n) (m) where
  heap : Vector (Node n m) m
  root : Pointer m
```

We build up our BDD representation using two lower-level notions: *pointers* and *nodes*. Given a natural number m, terms of type Pointer m represents pointers which are either one of two distinguished *terminal* pointers, labeled by the two Booleans true and false, or they are indices in an array of size m.

Pointers that point to an array index are terms node i where i : Fin m. The use of Fin m instead of, say, Nat means that our pointers are safe by construction: they can only represent valid (in-bounds) pointers for arrays of size m. The two terminal pointers are akin to the NULL pointer in the C programming language (and others) in that they point nowhere and cannot be dereferenced, but unlike NULL, our terminals carry data—a single Boolean stored in the pointer itself, that says which terminal it is.

A term N of type Node n m represents one nonterminal node in a BDD with input size n and m nonterminals in total. The field N.var is the variable index of the node, corresponding to the var function from definition 8. The N.low and N.high fields represent the low and high edges of node.

Note the odd detail that Nodes are indexed by m, the number of nodes in an array. This is required because the low and high fields of a node are (valid) pointers, which are indexed by the maximum node index.

Lastly, a term B of type Bdd n m consists of two fields: B.heap and B.root. The former is an array of m nodes (which is represented by the type Vector (Node n m) m), and the latter is a pointer, either a terminal or a pointer to one of the nodes in B.heap.

Any vector M : Vector (Node n m) m of m nodes of type Node n m induces a directed graph $\langle V, E\rangle$, wherein the edge relation E is given by the following Edge M relation over Pointer m:

```
inductive Edge (M : Vector (Node n m) m) : Pointer m → Pointer m → Prop where
  | low  : M[j].low  = p → Edge M (node j) p
  | high : M[j].high = p → Edge M (node j) p
```

In words,

- The vertices of the graph are *pointers*, namely they are the set of all terms of type `Pointer m`. Put differently, $V = \mathbb{B} \oplus [m]$.
- There is an edge in the graph from pointer `p` to pointer `q` if and only if `p` is a nonterminal pointer `node j` pointing to the node `M[j]` for some index `j : Fin m`, and `q` is either the low pointer or the high pointer of `M[j]`.

So we read a term `Edge M p q` as a proof of the statement: "there is an edge from p to q in the graph induced by M".

We recover something that resembles our graph-based definition of BDDs in definition 8 from our concrete representation in Lean by thinking of a term `B : Bdd n m` as the graph over the `Pointer m` type induced by the vector `B.heap`, along with a distinguished pointer `B.root`.

For example, we can represent the BDD denoting the ternary majority function (as seen in fig. 1.1) as the following `Bdd` term:

```
example : Bdd 3 4 :=
  { heap := ⟨#[{var := 0, low := node 1,         high := node 2},
               {var := 1, low := terminal false, high := node 3},
               {var := 1, low := node 3,         high := terminal true},
               {var := 2, low := terminal false, high := terminal true}], rfl⟩
    root := node 0 }
```

There are, however, two differences with respect to definition 8:

- The graphs induced by `Bdd n m` terms are not guaranteed to be acyclic. Recall that we relied on the acyclicity assumption, for one, to justify the that our BDD denotation function is well-defined (see definition 11), but our concrete representation may even yield graphs with self loops! For example, the following `Bdd` term induces a graph with one node, such that both the low and high edges of that node point back to itself:

  ```
  example : Bdd 1 1 := ⟨Vector.singleton ⟨0, .node 0, .node 0⟩, .node 0⟩
  ```

  We could leverage Lean's rich type system to encode the acyclicity requirement in the raw `Bdd n m` type, but it would make the definition more cumbersome, and provide little benefit in return: recall that we are building up towards *ordered* BDDs, which are already guaranteed to be acyclic by virtue of being ordered.

- The graphs induced by `Bdd n m` terms always have exactly two terminal nodes, one for each Boolean, whereas in definition 8, to keep the definition as general as possible, we allowed any number of terminal nodes for either Boolean. This does not hinder the expressivity of our implementation in practice, because additional terminal nodes never arise in Reduced BDDs anyway. On the other hand it allows us to slightly reduce the memory footprint of a BDD by representing terminal node implicitly, without allocating room for them in the BDD heap. In particular, we can represent BDDs whose root is a terminal with a mere terminal pointer and a completely empty heap.

  ```
  example : Bdd n 0 := ⟨Vector.emptyWithCapacity 0, .terminal true⟩
  ```

By taking the reflexive transitive closure of the `Edge` relation, we obtain a reachability relation, which will come in handy in many of the following constructions:

```
def Reachable (M : Vector (Node n m) m) := Relation.ReflTransGen (Edge M)
```

Note that the `Reachable` relation is indexed by a given vector of nodes, which induces the underlying `Edge` relation.

Also note that the heap of a `Bdd` may contain nodes that are not `Reachable` from the root pointer. Intuitively, only nodes that are reachable from the root of a BDD are relevant for its denotation. The possibility of having irrelevant nodes in our heap is a feature, not a bug, as it allows us to consider and type proper sub-`Bdd`s of a given `Bdd`, wherein the root of the original `Bdd` is not reachable from the roots of its sub-`Bdd`s, for example. The fact that we can give sub-`Bdd`s the same type as the whole `Bdd` allows us to define functions and prove statements about `Bdd`s using recursion.

We formalize the notion of a relevant pointer with respect to a given BDD with the following definition:

Definition 33 (B-Relevant Pointer)**.** *We say that a pointer p is* relevant *with respect to a BDD B (or B-relevant in short) if p is reachable from the root of B in the graph induced by the heap of B.*

```
def RelevantPointer (B : Bdd n m) := { q // Reachable B.heap B.root q }
```

Here again we use Lean's subtype notation, namely we define `RelevantPointer B` as a subtype of `Pointer m`, where `Pointer m` is implied and automatically inferred by Lean.

Additionally, we define *relevant edges* for a given BDD as edges between relevant pointers, by lifting the `Edge` relation to `RelevantPointers`:

```
def RelevantEdge (B : Bdd n m) (p q : B.RelevantPointer) := Edge B.heap p.1 q.1
```

Given an `Edge` whose starting point is reachable, we can easily obtain a `RelevantEdge` with the same endpoints:

```
lemma relevantEdge_of_edge_of_reachable {B : Bdd n m}
    (e : Edge B.heap p q) (hp : Reachable B.heap B.root p) :
  RelevantEdge B ⟨p, hp⟩ ⟨q, .tail hp e⟩ := e
```

2.2.2 Properties

Ordered BDDs

We proceed with formalizing the notion of an *ordered* BDD. Intuitively, a BDD is ordered if all edges in the graph respect the variable ordering in the sense that edges go from a lower variable index to a higher variable index. Of course, edges as we defined them in the previous section are a relation between *pointers*, not between nodes, and pointers include terminals, which do not have a pertinent variable index.

To paper over this distinction, we define the function `toVar`, which maps pointers to an extended domain of variable indices, in which terminals get a special value distinct of all the "usual" variable indices associated with BDD nodes:

```
def toVar (M : Vec (Node n m) m) : Pointer m → Fin n.succ
  | terminal _ => n
  | node j     => M[j].var
```

The function toVar naturally requires a heap in which node pointers are resolved. Since all nodes have variable indices up to but not including n, we can safely assign n itself as the pseudo variable index of terminals.

Using toVar, we can define a relation on pointers p, q that is a precondition for the existence of an edge from p to q in an ordered BDD. We call this relation MayPrecede:

```
def MayPrecede (M : Vec (Node n m) m) (p q : Pointer m) := toVar M p < toVar M q
```

Similarly to toVar, the MayPrecede relation is also indexed by a heap M, in which node pointers are resolved.

As a sanity check, we prove a little lemma which says that terminal pointers never precede any pointer:

```
lemma not_terminal_MayPrecede : ¬ MayPrecede M (terminal b) p := by
  cases p with
  | terminal _ => simp [MayPrecede]
  | node     j => exact not_lt.mpr (Fin.le_last _)
```

In our Lean definition of *ordered* BDDs, we want to require all edges in the BDD's induced graph to respect the MayPrecede relation. Of course, only *relevant* edges matters. For convenience, we lift MayPrecede from a relation on m-pointers to a relation on B-relevant pointers similarly to how we defined the RelevantEdge relation:

```
def RelevantMayPrecede (B : Bdd n m) (p q : B.RelevantPointer) :=
  MayPrecede B.heap p.1 q.1
```

We can now readily define the orderedness predicate Ordered, which holds for BDDs in which the existence of a *relevant* edge from pointer p to q implies that p *may precede* q:

```
def Ordered (B : Bdd n m) := Subrelation (RelevantEdge B) (RelevantMayPrecede B)
```

We define Ordered as a constraint on all relevant edges. This makes it quite trivial to formalize the proof of lemma 17:

```
lemma Bdd.ordered_of_reachable {O : OBdd n m} :
    Reachable O.1.heap O.1.root p → Ordered ⟨O.1.heap, p⟩ := fun hp ⟨_, hx⟩ _ _ ↦
  O.2 (relevantEdge_of_edge_of_reachable (by simp_all) (.trans hp hx))
```

The Ordered property is of utmost importance, since it is needed to establish BDDs as a *canonical* representation of Boolean functions. In addition, OBdds are much more well behaved than generic Bdds, since the graph of a OBdd is always well-founded, while the graphs of arbitrary Bdds may in principle contain loops. This well-foundedness allows us to perform well-founded recursion on OBdds in Lean, which is an essential tool for defining functions and proving theorems.

To formally prove well-foundedness, we first lift the Edge relation to a corresponding relation between OBdds:

Definition 34 (OBDD Edge)**.** *Given OBDDs O and U, we say that there's an* edge *from O to U if O and U share the same heap M and there's an edge from the root of O to the root of U in the graph induced by M.*

```
def OEdge (O U : OBdd n m) := O.1.heap = U.1.heap ∧ Edge O.1.heap O.1.root U.1.root
```

In our proof of the following well-foundedness lemma 38, we rely on the following three facts, which correspond to lemmas `Subrelation.wf`, `InvImage.wf` and `Nat.lt_wfRel` in Lean, respectively:

Fact 35. *A subrelation of a well-founded relation is well-founded.*

Fact 36. *The inverse image of a well-founded relation by any function is well-founded.*

Fact 37. *The $<$ relation on the natural numbers is well-founded.*

Lemma 38. *The OBDD edge relation is well-founded.*

Proof. We define the function f from OBDDs to the natural numbers as the function that maps an OBDD O to `toVar` of O's root with respect to O's heap. For all OBDDs O and U, if there is an edge from O to U then by orderedness we have $f(O) < f(U)$. Thus the edge relation is a subrelation of the inverse image of $<$ by f, and from facts 35, 36 and 37 we get that the edge relation is well-founded. $\square$

The OBDD edge relation is also *converse* well-founded, which turns out to be the more useful direction for recursion purposes, where we usually want to prove a statement about the root of an OBDD while assuming the statement holds for its sub-OBDDs.

Lemma 39. *The OBDD edge relation is converse well-founded.*

Proof. Similar to the last proof, except with f replaced with the function $g(O) = n - f(O)$, where n is the input size of O. $\square$

This well-foundedness of `OBdd` edges allows us to formalize the translation of OBDDs to *decision trees* (definition 18) via well-founded recursion. We define decision trees in Lean as follows:

```
inductive DecisionTree n where
  | leaf   : Bool  → DecisionTree _
  | branch : Fin n → DecisionTree n → DecisionTree n → DecisionTree n
```

The `DecisionTree` is indexed by a natural number `n`, which upper-bounds the indices that may appear on the "branches" of the tree, analogously to the input size `n` of an `OBdd n m` term.

We can *evaluate* a given decision tree, with respect to some input input vector of `n` Booleans to obtain an output Boolean, by following the branches of the tree according to the values of the input Booleans. Formally:

```
def DecisionTree.evaluate : DecisionTree n → Vec Bool n → Bool
  | leaf b      , _ => b
  | branch j l h, v => if v[j] then h.evaluate v else l.evaluate v
```

Next, we would like to formalize the translation from OBDDs to decision trees that we defined in definition 19. Naturally, we shall define this translation recursively. To facilitate recursion on OBDDs, we introduce a couple of helper definitions first:

```
def Bdd.low (B : Bdd n m) : B.root = node j → Bdd n m
  | _ => {heap := B.heap, root := B.heap[j].low}

def Bdd.high (B : Bdd n m) : B.root = node j → Bdd n m
  | _ => {heap := B.heap, root := B.heap[j].high}
```

We also prove a pair lemmas that allows us to conclude that low(B) and high(B) are ordered given that B is ordered.

Lemma 40. *If B is ordered, then also* low(B) *and* high(B) *are ordered.*

Proof. Immediate from lemma 17 and the fact that the roots of low(B) and high(B) are reachable from the root of B. □

```
lemma Bdd.high_ordered {B : Bdd n m} (h : B.root = node j) : B.Ordered → (B.high h).Ordered
lemma Bdd.low_ordered  {B : Bdd n m} (h : B.root = node j) : B.Ordered → (B.low  h).Ordered
```

Using these lemmas, we lift the definition of the BDDs low(B) and high(B) for BDD B to OBDDs low(O) and high(O) for OBDD O:

```
def OBdd.high (O : OBdd n m) : O.1.root = node j → OBdd n m
  | h => ⟨O.1.high h, Bdd.high_ordered h O.2⟩

def OBdd.low (O : OBdd n m) : O.1.root = node j → OBdd n m
  | h => ⟨O.1.low h, Bdd.low_ordered h O.2⟩
```

The following lemmas are also trivial, they assert that O and low(O) stand in the well-founded OBDD edge relation (and likewise for high(O)):

```
lemma oedge_of_low  {h : O.1.root = node j} : OEdge O (O.low  h)
lemma oedge_of_high {h : O.1.root = node j} : OEdge O (O.high h)
```

To instruct Lean to try and use the above lemmas to convince itself that recursive definitions are correct if they go from O to low(O) and/or high(O), we extend the built-in tactic `decreasing_trivial`, which Lean employs to automatically establish that a given recursive call follows a well-founded order:

```
macro_rules | `(tactic| decreasing_trivial) => `(tactic| exact oedge_of_low)
macro_rules | `(tactic| decreasing_trivial) => `(tactic| exact oedge_of_high)
```

With that, we are fully equipped to provide our first simple recursive function on OBDDs:

```
def OBdd.toTree (O : OBdd n m) : DecisionTree n :=
  match h : O.1.root with
  | terminal b => .leaf b
  | node j     => .branch O.1.heap[j].var (toTree (O.low h)) (toTree (O.high h))
termination_by O
```

We can compose the evaluation function defined above for decision trees with the `toTree` mapping of OBDDs to decision trees, to obtain an evaluation function for OBDDs:

```
def OBdd.evaluate : OBdd n m → BoolFun n := DecisionTree.evaluate ∘ OBdd.toTree
```

The `OBdd.evaluate` function yields a Boolean function which corresponds to the denotation of a BDD (definition 11). Note that constructing the decision tree corresponding to a BDD is a form of decompression, which can potentially be quite costly. In section 2.2.3 we show a simple and efficient BDD evaluation algorithm, which walks the BDD graph directly instead of constructing an intermediary decision tree.

BDD Size

We also leverage well-founded recursion on OBDDs to implement a decision procedure for reachability from the root of an OBDD, which simply walks down the BDD graph from its root and sees if it encounters the pointer in question.

```
instance instDecidableReachable (O : OBdd n m) :
  DecidablePred (Reachable O.1.heap O.1.root)
```

With the above instance in place, Lean can infer that the type of pointers reachable from the root of an OBDD is a `Fintype`, which means that it is a *finite* type equipped with a procedure for *producing* its finitely many inhabitants. Lean is able to establish that since the type of all pointers `Pointer m` is a `Fintype`, and the pointers reachable from an OBDD's root make up a subset of that `Fintype`, which is defined by a decidable property. (In Lean, there is also a notion of `Finite` types, which similarly express finiteness, but do not provide a way to compute the terms of the finite type.)

This allows us to formalize our notion of the size of a BDD from definition 10 for OBDDs, as the cardinality (number of elements) in the finite type of nonterminal pointers reachable from the root of an OBDD:

```
def OBdd.size {n m} (O : OBdd n m) :=
  Fintype.card { j // Reachable O.1.heap O.1.root (.node j) }
```

Note that this definition is fully computable—we can use it to calculate the size of a given OBDD. The computational content of this definition comes from our `instDecidableReachable` instance: to compute `OBdd.size` of a given OBDD, Lean goes over all indices `j`, checks if `node j` is reachable from the OBDD root using `instDecidableReachable`, and counts how may such pointers it found. This definition is convenient for formalized reasoning, but since it decides for each index whether it is reachable or not by traversing the entire BDD graph, it is clearly not computationally efficient. To efficiently obtain the size of large OBDDs, we provide an alternative implementation that traverses the BDD graph only once in file `Size.lean`:

```
def Size.size : OBdd n m → Nat := List.length ∘ Collect.collect
```

The `Collect.collect` function, which we describe in more detail in section 2.2.5, performs a DFS traversal to produce a list of all reachable nodes. All that remain for our `Size.size` to do is to measure the length of the resulting list. Then the correctness of `Size.size`, which is stated in the following lemma`Size.size_spec`, is obtained as a consequence of the correctness of `Collect.collect`:

```
lemma size_spec {O : OBdd n m} : Size.size O = OBdd.size O
```

We also formalize and prove our lemma 16, which gives an upper bound on the size of OBDDs:

```
lemma size_le {O : OBdd n m} : size O ≤ 2 ^ n - 1
```

Reduced BDDs

The `toTree` function we saw in section 2.2.2 is also instrumental in defining the notion of a *reduced* OBDD via the BDD similarity relation, as we saw in definition 24 and 20.

Formally, we first define a "heterogeneous" similarity relation, `HSimilar`, which relates OBDDs that may differ in their heap size, and thus inhabit different types. Homogeneous similarity, which we call `Similar`, is then obtained as a special case of the heterogeneous definition:

```
def OBdd.HSimilar (O : OBdd n m) (U : OBdd n m') := O.toTree = U.toTree

def OBdd.Similar : OBdd n m → OBdd n m → Prop := HSimilar
```

For convenience, we also define a corresponding similarity relation restricted to relevant pointers for a given OBDD:

```
def OBdd.SimilarRP (O : OBdd n m) (p q : O.1.RelevantPointer) :=
  Similar ⟨{heap := O.1.heap, root := p.1}, ordered_of_reachable p.2⟩
          ⟨{heap := O.1.heap, root := q.1}, ordered_of_reachable q.2⟩
```

Next, we formalize the notion of redundancy from definition 23:

```
inductive Pointer.Redundant (M : Vec (Node n m) m) : Pointer m → Prop where
  | red : M[j].low = M[j].high → Redundant M (node j)
```

For convenience, we also define a predicate `NoRedundancy` that holds for BDDs for which no relevant pointer is redundant:

```
def Bdd.NoRedundancy (B : Bdd n m) := ∀ (p : B.RelevantPointer), ¬ Redundant B.heap p.1
```

The formal definitions of similarity and redundancy suffice for formalizing definition 24:

```
def OBdd.Reduced (O : OBdd n m) : Prop
  -- No redundant pointers.
  := NoRedundancy O.1
  -- Similarity implies pointer-equality.
   ∧ Subrelation (SimilarRP O) (InvImage Eq Subtype.val)
```

With the definition of the `Reduced` predicate in place, we have now covered all of the ingredients that take part in the `BDD` type definition that we saw in section 2.2.1.

2.2.3 BDD Evaluation

In section 2.1.1, we saw that the interface function `BDD.denotation` is implemented in terms of two other functions, `BDD.lift` and `BDD.evaluate`:

```
def denotation (B : BDD) (h : B.nvars ≤ n) : BoolFunc n := (B.lift h).evaluate
```

The `BDD.evaluate` function is in turn a wrapper around a function on the underlying `OBdd`, called `Evaluate.evaluate`:

```
abbrev evaluate (B : BDD) : BoolFunc B.nvars := Evaluate.evaluate B.obdd
```

The `Evaluate.evaluate` function implements the BDD evaluation algorithm, which produces the output of the denotation of a BDD for a given input by simply walking down the BDD from the root down to a terminal node, taking low or high edges based on the values of the input Booleans:

```
def Evaluate.evaluate (O : OBdd n m) : BoolFunc n := fun I ↦
  match h : O.1.root with
  | .terminal b => b
  | .node j => if I[O.1.heap[j].var] then evaluate (O.high h) I else evaluate (O.low h) I
termination_by O
```

We show that this definition is equivalent to the `DecisionTree`-based implementation of `OBdd.evaluate` which we saw in section 2.2.2, via a simple recursive proof.

```
lemma Evaluate.evaluate_eq_treeEvaluate : Evaluate.evaluate O = OBdd.evaluate O
```

This permits us to continue reasoning about BDD evaluation using `OBdd.evaluate` when that is more convenient, while at runtime our library always uses the more efficient `Evaluate.evaluate`.

The `BDD.lift` function is similarly a wrapper around `Lift.lift`, which implements the BDD lifting discussed in section 2.1.1:

```
def lift (h : n ≤ n') (B : Bdd n m) : Bdd n' m :=
  ⟨ Vector.map
      (fun N ↦ ⟨⟨N.var.1, Fin.val_lt_of_le N.var h⟩, N.low, N.high⟩)
      B.heap,
    B.root
  ⟩
```

The `Lift.lift` function essentially leaves its input unchanged, adjusting only the proofs that the variable indices of all of the nodes in the BDD heap are valid. Although lifting seems like a no-op, we must nevertheless prove to Lean that it preserves the BDD properties that we care about. Namely, we prove that lifting reduced ordered BDDs produces reduced ordered BDDs, starting from orderedness:

```
lemma lift_ordered {h : n ≤ n'} {B : Bdd n m} : B.Ordered → (lift h B).Ordered
```

The `lift_ordered` allows us to define a lift operation on OBDDs that we call `olift`. We further prove that `olift` preserves the property of being reduced:

```
def olift (h : n ≤ n') (O : OBdd n m) : OBdd n' m := ⟨(lift h O.1), lift_ordered O.2⟩

lemma olift_reduced {h : n ≤ n'} {O : OBdd n m} : O.Reduced → (olift h O).Reduced
```

Lastly we show that our lift operation has the expected behavior with regards to the denotation of the lifted BDD, namely that evaluating the resulting BDD with respect to some input is the same as evaluating the original BDD with respect to the appropriate prefix of that input:

```
lemma olift_evaluate {h : n ≤ n'} {O : OBdd n m} {I : Vector Bool n'} :
  (olift h O).evaluate I = O.evaluate (Vector.cast (by simpa) (I.take n))
```

We use `olift_evaluate` to prove the `denotation_independentOf_of_geq_nvars` lemma which we saw in section 2.1.1.

2.2.4 Semantic Equivalence and Similarity

Next we take a closer look at `BDD.instDecidableSemanticEquiv`, the `BDD` semantic equivalence decision procedure that we presented in section 2.1.2. The general strategy was sketched in section 1.2—we want to rely on the *canonicity* of BDDs (theorem 25), which allows us to infer that BDDs are equivalent whenever their concrete representation agrees.

Our formal proof of BDD canonicity in Lean relies on a few helper lemmas and definitions. For convenience, we define a function `OBdd.var`, which returns the variable index at the root of a given OBDD, or the OBDD's input size in case its root is a terminal:

```
def Bdd.var  (B : Bdd  n m) : Fin n.succ := B.root.toVar B.heap

def OBdd.var (O : OBdd n m) : Nat := O.1.var
```

Next, we prove a couple of lemmas about how `OBdd.var` increases as we go down a given OBDD, which are easily obtained from the orderedness of OBDDs:

```
lemma OBdd.var_lt_high_var {O : OBdd n m} {h : O.1.root = node j} : O.var < (O.high h).var

lemma OBdd.var_lt_low_var  {O : OBdd n m} {h : O.1.root = node j} : O.var < (O.low h).var
```

Using these lemmas along with OBDD well-founded recursion, we prove the following lemma which states that the denotation of an OBDD `O` is independent of any variable index below `O.var`:

```
lemma OBdd.independentOf_lt_root (O : OBdd n m) (i : Fin O.var) :
    IndependentOf (O.evaluate) ⟨i.1, Fin.val_lt_of_le i (Fin.is_le _)⟩ := by
  cases h : O.1.root with
  | terminal _ => simp [evaluate_terminal' h]
  | node j =>
    intro b I
    rw [evaluate_node'' h]
    simp only
    rcases i with ⟨i, hi⟩
    congr 1
    · simp only [eq_iff_iff, Bool.coe_iff_coe]
      symm
      apply Vector.getElem_set_ne _ _ (Nat.ne_of_lt (by simp_all))
    · exact (independentOf_lt_root (O.high h) ⟨i, .trans hi var_lt_high_var⟩) b I
    · exact (independentOf_lt_root (O.low  h) ⟨i, .trans hi var_lt_low_var⟩) b I
termination_by O
```

We use the `independentOf_lt_root` lemma to formally prove our canonicity theorem, along the lines of the proof we gave for theorem 25. Given two reduced OBDDs, we proceed by case distinction on the nature of their root pointers, followed by appealing to canonicity of their sub-BDDs via well-founded recursion:

```
theorem OBdd.Canonicity {O : OBdd n m} {U : OBdd n m'} (ho : O.Reduced) (hu : U.Reduced) :
    O.evaluate = U.evaluate → O.HSimilar U := by
  intro h
  cases hro : O.1.root with
  | terminal b =>
    cases hru : U.1.root with
    | terminal c => ...
    | node i =>
      rw [evaluate_terminal' hro] at h
      have : (U.high hru).evaluate = (U.low hru).evaluate := ...
      absurd hu
      apply not_reduced_of_sim_high_low hru
      apply Canonicity (high_reduced hu) (low_reduced hu) this
  | node j =>
    cases hru : U.1.root with
    | terminal c =>
      ...
      apply not_reduced_of_sim_high_low hro
      apply Canonicity (high_reduced ho) (low_reduced ho) this
    | node i =>
      ...
      have same_var : O.1.heap[j].var = U.1.heap[i].var := by
```

```
        apply eq_iff_le_not_lt.mpr
        constructor
        · apply le_of_not_lt
          intro contra
          have := independentOf_lt_root O ⟨U.1.heap[i].var.1, ...⟩
          ...
        · intro contra
          have := independentOf_lt_root U ⟨O.1.heap[j].var.1, ...⟩
          ...
      constructor
      · exact same_var
      · constructor
        · apply Canonicity (low_reduced  ho) (low_reduced  hu) ...
        · apply Canonicity (high_reduced ho) (high_reduced hu) ...
termination_by O.size + U.size
decreasing_by ...
```

Note that the well-founded recursion in this proof requires a different decreasing measure than our usual well-founded edge relation on OBDDs, since we apply the theorem recursively also in cases in which the first argument in the recursive call is not a sub-OBDD of O, but rather it is a sub-OBDD of U, and vice-versa. We thus use the sum of the sizes of the two input OBDDs as our decreasing measure, and prove that all recursive calls indeed respect this well-founded relation.

We also prove a Canonicity_reverse result, which states that similar ROBDDs have the same denotation. If fact, this holds for any two OBDDs, not necessarily reduced. Unlike Canonicity, the reverse statement is trivial to prove given our definitions:

```
theorem OBdd.Canonicity_reverse {O : OBdd n m} {U : OBdd n m'}:
    O.HSimilar U → O.evaluate = U.evaluate := by
  simp_all [evaluate, Function.comp_apply, Similar, HSimilar]
```

Our next steps towards instDecidableSemanticEquiv is to lift the HSimilar relation, which we have defined for OBDDs of a given input size, to similarity relation on our abstract BDD type. To do so, we use the OBDD lifting operation to obtain OBDDs with the same input size given any two BDDs:

```
private def BDD.Similar (B : BDD) (B' : BDD) :=
  OBdd.HSimilar
    (Lift.olift (Nat.le_max_left  ..) B.obdd)
    (Lift.olift (Nat.le_max_right ..) B'.obdd)
```

We use our canonicity theorems to show that for ROBDDs, similarity is equivalent to semantic equivalence:

```
private theorem SemanticEquiv_iff_Similar {B C : BDD} :
    B.SemanticEquiv C ↔ B.Similar C := ⟨l_to_r, r_to_l⟩ where
  l_to_r h := by
    simp only [SemanticEquiv, denotation, Evaluate.evaluate_eq_treeEvaluate] at h
    exact Canonicity (Lift.olift_reduced B.hred) (Lift.olift_reduced C.hred) h
  r_to_l h := by
    simp only [SemanticEquiv, denotation, Evaluate.evaluate_eq_treeEvaluate]
    exact Canonicity_reverse h
```

We appeal to exactly this equivalence between semantic and similarity for ROBDDs in our implementation of `instDecidableSemanticEquiv`:

```
instance instDecidableSemanticEquiv : DecidableRel SemanticEquiv
  | _, _ => decidable_of_iff' _ SemanticEquiv_iff_Similar
```

This definition reads as follows: to decide whether two `BDD` are semantically equivalent, decide whether they are similar, which suffices since the two relations coincide.

The last piece of the puzzle is providing a decision procedure for the `BDD.Similar` relation for ROBDDs. We can, of course, check if two OBDDs are similar by simply translating them to decision trees and comparing the resulting trees for equality. However, we want to avoid the potentially high performance cost that may incur. In particular, when deciding whether a given ROBDD denotes a contradiction (or a tautology), we need to compare an arbitrarily large ROBDD to an ROBDD of size 0, since by canonicity we know that contradictions (and tautologies) are represented by ROBDDs whose root is a terminal. In those cases, we most certainly do not want to unfold the arbitrarily large BDD to a decision tree before comparing its root to a terminal. To provide good performance guarantees, we implement algorithm 1, which decides whether two ROBDDs are similar in time proportional to the size of the smaller among the two ROBDDs.

We implement algorithm 1 in file `Sim.lean`. The crucial aspect of this algorithm is that by keeping a record of similar sub-BDDs in the two hash tables H_{OU} and H_{UO}, we get at most one recursive call for each node in each of the two ROBDDs. Since in each call we perform checks that take constant time, the algorithm has time complexity $O(min\{\text{size}(O), \text{size}(U)\})$ in total. Note that in line *, we are justified in concluding that $O \sim U$ by virtue of having already establishing the following:

$$\begin{aligned}\text{var}(\text{root}(O)) &= \text{var}(\text{root}(U))\\ O[\text{low}(\text{root}(O))] &\sim U[\text{low}(\text{root}(U))]\\ O[\text{high}(\text{root}(O))] &\sim U[\text{high}(\text{root}(U))]\end{aligned}$$

Also note that the recursive invocations of the algorithm destructively mutate the hash tables. This is desirable, because it means that if, for example, a node v is reachable both from low(root(O)) and from high(root(O)), then by the time we encounter v when going down the high edge of root(O), we already have a record for v in our hash tables from our previous descent down the low edge of root(O).

This kind of stateful computation, in which a side-effect of one part of the computation, in addition to producing some result, also affects another part by mutating some commonly accessible state, is ubiquitous in BDD algorithms—both in the literature about these algorithms, and in their implementations in practice. Indeed, many BDD implementations are written in *imperative* programming languages which allow such side-effects via direct memory manipulation. For example, the popular BDD package CUDD and CacBDD are implemented as low-level C and C++ libraries, respectively [Som98; LSX13]. (A notable exception is Markus Triska's BDD implementation in Prolog [Tri16], but even that implementation relies heavily on mutating *variable attributes*.)

To implement such destructive mutation in the purely functional context of Lean, we embed our algorithm in the `StateM` monad, which allows for computations that access and mutate a given state object.

We define the type of states of our algorithm as a structure that holds the two hash tables, as well as two invariant proofs that guarantee that the hash tables are symmetric and that any mapping they

```
Input: Two ROBDDs $O$ and $U$
Output: true if $O \sim U$, false if $O \nsim U$
$H_{OU} \leftarrow$ empty hash table mapping nodes in $O$ to nodes in $U$
$H_{UO} \leftarrow$ empty hash table mapping nodes in $U$ to nodes in $O$
def sim_helper($O$,$U$):
    if root($O$) is terminal then
        if root($U$) is terminal then
            if val(root($O$)) = val(root($U$)) then
                return $O \sim U$
            else
                return $O \nsim U$
            end
        else
            return $O \nsim U$
        end
    else
        if root($U$) is terminal then
            return $O \nsim U$
        else
            if var(root($O$)) = var(root($U$)) then
                if $H_{OU}$[root($O$)] = $v$ for some node $v$ then
                    if $v$ = root($U$) then
                        return $O \sim U$
                    else
                        return $O \nsim U$
                    end
                else
                    if $H_{UO}$[root($U$)] = $v$ for some pointer $v$ then
                        return $O \nsim U$
                    else
                        if sim_helper($O$[low(root($O$))], $U$[low(root($U$))]) then
                            if sim_helper($O$[high(root($O$))], $U$[high(root($U$))]) then
                                $H_{OU}$[root($O$)] $\leftarrow$ root($U$)
                                $H_{UO}$[root($U$)] $\leftarrow$ root($O$)
*                                return $O \sim U$
                            else
                                return $O \nsim U$
                            end
                        else
                            return $O \nsim U$
                        end
                    end
                end
            else
                return $O \nsim U$
            end
        end
    end
return sim_helper($O$,$U$)
```

Algorithm 1: ROBDD similarity decision procedure

include implies similarity of the corresponding sub-BDDs:

```
structure State (O : OBdd n m) (U : OBdd n m’) where
  lr : Std.HashMap (Fin m) (Fin m’)
  rl : Std.HashMap (Fin m’) (Fin m)
  hl : ∀ j j’,
    lr[j]? = some j’ → rl[j’]? = some j  ∧
    Reachable O.1.heap O.1.root (.node j) ∧
    ∃ hj : Bdd.Ordered ⟨O.1.heap, .node j⟩,
      ∃ hj’ : Bdd.Ordered ⟨U.1.heap, .node j’⟩,
        OBdd.HSimilar ⟨⟨O.1.heap, .node j⟩, hj⟩ ⟨⟨U.1.heap, .node j’⟩, hj’⟩
  hr : ∀ j j’,
    rl[j’]? = some j → lr[j]? = some j’ ∧ ...
```

The core of the algorithm is then implemented in a recursive helper function `sim_helper` that operates on two nodes, each reachable from the root of one of the ROBDDs that we want to compare, and decides whether the OBDDs rooted at these two nodes are similar. In order to be able to access and mutate our two hash tables, the `sim_helper` function works inside the `StateM` monad:

```
  def sim_helper
      (O : OBdd n m) (hO : OBdd.Reduced O)
      (U : OBdd n m’) (hU : OBdd.Reduced U)
      (p : Pointer m) (hpr : Reachable O.1.heap O.1.root p)
      (q : Pointer m’) (hqr :Reachable U.1.heap U.1.root q) :
    StateM
      (State O U)
      (Decidable
        (OBdd.HSimilar
          ⟨⟨O.1.heap, p⟩, Bdd.ordered_of_reachable hpr⟩
          ⟨⟨U.1.heap, q⟩, Bdd.ordered_of_reachable hqr⟩)) := do
    match hp : p with
    | .terminal b =>
      match hq : q with
      | .terminal b’ =>
        if hb : b = b’
        then return isTrue (by simpa [...])
        else return isFalse (by simpa [...])
      | .node j’ => return isFalse (by simp [...])
    | .node j =>
      match hq : q with
      | .terminal b’ => return isFalse (by simp [...])
      | .node j’ =>
        if hv : O.1.heap[j].var = U.1.heap[j’].var
        then ...
        else return isFalse (by simp_all [...])
```

Lastly, we wrap the `sim_helper` function with a definition that creates an initial state object and kick-starts the algorithm at the roots of two given ROBDDs. Note that in order to initialize the state for the algorithm, we must provide two hash tables and prove that they satisfy the required invariants. Fortunately, since our invariants are statements about records in the hash tables, proving that they hold for the initial empty hash tables is trivial:

```
instance Sim.instDecidableRobddHSimilar (O : OBdd n m) (U : OBdd n m')
    (hO : O.Reduced) (hU : U.Reduced) : Decidable (O.HSimilar U) :=
  ((sim_helper O hO U hU O.1.root .refl U.1.root .refl)
    ⟨ Std.HashMap.emptyWithCapacity 0,
      Std.HashMap.emptyWithCapacity 0,
      by simp, by simp ⟩).1
```

2.2.5 BDD Construction Algorithms

In section 2.1.3 we presented and explained the use of the functions our library provides for constructing BDDs, from `BDD.const` and `BDD.var` through `BDD.restrict` and `BDD.apply` (and its derivatives, `and`, `or`, `not`, etc.) to `BDD.relabel` and `BDD.foralls`. In this section, we examine their implementation and verification.

Basic Building Blocks

For `BDD.const`, we explicitly construct a `BDD` term with an empty heap and a terminal pointer at the root. The `BDD` type also requires us to show that the BDD we construct is ordered and reduced, which we do with the help of two useful lemmas, `Bdd.ordered_of_terminal` and `Bdd.reduced_of_terminal`, which state that BDDs whose root is a terminal are ordered and reduced:

```
lemma Bdd.ordered_of_terminal : Bdd.Ordered  ⟨M, terminal b⟩
lemma Bdd.reduced_of_terminal : Bdd.Reduced ⟨⟨M, terminal b⟩, o⟩

def const (b : Bool) : BDD :=
  { nvars := 0,
    nheap := 0,
    obdd  := ⟨⟨Vector.emptyWithCapacity 0, .terminal b⟩, Bdd.ordered_of_terminal⟩,
    hred  := Bdd.reduced_of_terminal
  }
```

Given this definition of `BDD.const`, the proofs of the two interface lemmas that characterize it, `BDD.const_denotation` and `BDD.const_nvars`, are both trivial.

For `BDD.var`, we similarly construct a `BDD` explicitly, this time with a heap consisting of a single node, corresponding to a single input variable. We use dedicated helper lemmas `var_ordered` and `var_reduced` to establish that the resulting BDD is ordered and reduced:

```
def var (n : Nat) : BDD :=
  { nvars := n + 1,
    nheap := 1,
    obdd  := ⟨⟨Vector.singleton ⟨n, terminal false, terminal true⟩, node 0⟩, var_ordered⟩,
    hred  := var_reduced
  }
```

Apply

Next, we present the implementation of the `BDD.apply`, which is the core facility for constructing BDDs (beyond the simplest cases, which `BDD.const` and `BDD.var` cover). Recall from section 2.1.3 that the purpose of `BDD.apply` is to combine two ROBDDs $O, U \in BDD_n$ and produce a new

ROBDD $W \in BDD_n$ such that $D(W) = D(O) \bullet D(U)$, for some binary Boolean operator $\cdot \bullet \cdot$. By $D(W) = D(O) \bullet D(U)$ we mean that for all $x \in \mathbb{B}^n$, $D(W)(x) = D(O)(x) \bullet D(U)(x)$.

Our implementation follows the *Apply* algorithm which Bryant describes in [Bry86]. In [Knu09], Knuth describes a more sophisticated variant of this algorithm, which uses breadth-first, rather than depth-first, traversal of the two BDD graphs, and produces an output OBDD that is also reduced. Knuth dubs it "the most important algorithm on binary decision diagrams". In [Bry18], Bryant also provides a variation of his original *Apply* algorithm which directly produces reduced OBDDs. Our implementation is close to Bryant's original *Apply* algorithm, with the twist that our implementation in Lean is accompanied by a formal correctness proof.

We implement `BDD.apply` by composing two subroutines, `Apply.oapply` and `Reduce.oreduce`. The former produces an OBDD with the correct denotation, and then the latter is used to reduce the resulting OBDD to a ROBDD.

We provide pseudo-code for the variant of the *Apply* algorithm that we implement in `Apply.oapply` as algorithm 2. Similarly to algorithm 1, algorithm 2 starts with an initialization step, which sets up an initial state, followed by a recursive execution step, which accesses and mutates the state while computing the output BDD.

The state of algorithm 2 consists of an array of BDD nodes V and a hash table H, both initially empty. During execution of the algorithm, we populate V with BDD nodes, and finally return it as the heap of the output BDD, paired with a corresponding root pointer. We refer to the number of nodes V contains as $\mathrm{size}(V)$.

The hash table H maps pairs of input nodes (one from O and one from U) to output nodes. It is a memoization cache that we use to avoid duplicate recursive calls to the helper function apply_helper.

Note that apply_helper contains 10 different possible recursive invocations of itself, such that in each call to apply_helper we either reach none of these recursive invocations , or we reach exactly 2 of them. To prove that the algorithm terminates, it suffices to observe that in each of the 10 recursive call sites, we either descend from O to $\mathrm{low}(O)$ or to $\mathrm{high}(O)$, or we keep O fixed and descend from U to $\mathrm{low}(U)$ or to $\mathrm{high}(U)$. Therefore all recursive calls follow the well-founded edge relation on OBDDs, lifted to a well-founded lexicographic order on pairs of OBDDs $\langle O, U\rangle$.

To prove the correctness of algorithm 2, we identify the following invariant that the recursive helper function apply_helper respects:

Definition 41 (Apply Invariant). *Given two OBDDs O and U, a binary Boolean operator $\cdot \bullet \cdot$, and a state $\langle H, V\rangle$, the* Apply invariant *is the conjunction of the following conditions:*

1. *For all nodes V_j in V, if $\mathrm{low}(V_j)$ is a nonterminal pointer to some node V_k, then $k < j$; and likewise for $\mathrm{high}(V_j)$. In other words, all pointers between nodes in V go from higher indices in V to lower indices.*

2. *For all pointers o, u, w such that $H[o, u] = w$ we have:*

    (a) If w is a nonterminal pointer, then:

    i. w points to a node V_j at a valid index $j < \mathrm{size}(V)$, and

    ii. $\mathrm{var}(V_j) = \min\{\mathrm{var}(O[o]), \mathrm{var}(U[u])\}$

    (b) $O[o]$ is ordered.

    (c) $U[u]$ is ordered.

```
Input: Two OBDDs $O$ and $U$, and a binary Boolean operator $\cdot \bullet \cdot : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$
Output: OBDD $W$ with $D(W) = D(O) \bullet D(U)$
$V \leftarrow$ empty array of BDD nodes
$H \leftarrow$ empty hash table mapping pairs of input pointers to one output pointer
def push_node($O$, $U$, $i$, $l$, $h$):
    $s \leftarrow$ size of $V$
    Push a new node $V_s$ at the end of $V$
    $\mathrm{var}(V_s) \leftarrow i$, $\mathrm{low}(V_s) \leftarrow l$, $\mathrm{high}(V_s) \leftarrow h$
    $H[\mathrm{root}(O), \mathrm{root}(U)] \leftarrow s$
    return $s$;
def apply_helper($O$, $U$):
    if $H[\mathrm{root}(O), \mathrm{root}(U)] = p$ for some pointer $p$ then
        return $p$
    else
        if $\mathrm{root}(O)$ is terminal then
            if $\mathrm{root}(U)$ is terminal then
                return terminal with value $\mathrm{val}(\mathrm{root}(O)) \bullet \mathrm{val}(\mathrm{root}(U))$
            else
                $l \leftarrow$ apply_helper($O$, $\mathrm{low}(U)$)
                $h \leftarrow$ apply_helper($O$, $\mathrm{high}(U)$)
                return push_node($O$, $U$, $\mathrm{var}(U)$, $l$, $h$)
            end
        else
            if $\mathrm{root}(U)$ is terminal then
                $l \leftarrow$ apply_helper($\mathrm{low}(O)$, $U$)
                $h \leftarrow$ apply_helper($\mathrm{high}(O)$, $U$)
                return push_node($O$, $U$, $\mathrm{var}(O)$, $l$, $h$)
            else
                if $\mathrm{var}(O) < \mathrm{var}(U)$ then
                    $l \leftarrow$ apply_helper($\mathrm{low}(O)$, $U$)
                    $h \leftarrow$ apply_helper($\mathrm{high}(O)$, $U$)
                    return push_node($O$, $U$, $\mathrm{var}(O)$, $l$, $h$)
                else
                    if $\mathrm{var}(\mathrm{root}(O)) > \mathrm{var}(\mathrm{root}(U))$ then
                        $l \leftarrow$ apply_helper($O$, $\mathrm{low}(U)$)
                        $h \leftarrow$ apply_helper($O$, $\mathrm{high}(U)$)
                        return push_node($O$, $U$, $\mathrm{var}(U)$, $l$, $h$)
                    else
*1                      $l \leftarrow$ apply_helper($\mathrm{low}(O)$, $\mathrm{low}(U)$)
*2                      $h \leftarrow$ apply_helper($\mathrm{high}(O)$, $\mathrm{high}(U)$)
*3                      return push_node($O$, $U$, $\mathrm{var}(U)$, $l$, $h$)
                    end
                end
            end
        end
    end
$r \leftarrow$ apply_helper($O$, $U$)$\rangle$
return BDD $\langle V, r\rangle$ with heap $V$ and root $r$
```

Algorithm 2: Apply

(d) *The BDD $W = \langle V, w \rangle$ with heap V and root w is ordered, and $D(W) = D(O[o]) \bullet D(U[u])$.*

Theorem 42 (Correctness of apply_helper)**.** *If the Apply invariant holds for a given state $\langle V, H \rangle$ with respects to OBDDs O, U and operator $\cdot \bullet \cdot$, then after invoking* apply_helper(O, U) *and obtaining an output pointer r and a new state $\langle V', H' \rangle$, the following statements hold:*

1. *The Apply invariant holds also for the new state $\langle V', H' \rangle$.*
2. $H'[\text{root}(O), \text{root}(U)] = r$.
3. $\text{size}(V) \leq \text{size}(V')$.
4. *For all pointers o, u:*
   (a) *If $\langle o, u \rangle \notin H'$ (no value associated with the key $\langle o, u \rangle$ in H'), then also $\langle o, u \rangle \notin H$.*
   (b) *For all pointers w, if $H[o, u] = w$ then $H'[o, u] = w$.*
   (c) *If $\langle o, u \rangle \notin H$ but $\langle o, u \rangle \in H'$, then o is reachable from* $\text{root}(O)$ *and u is reachable from* $\text{root}(U)$.

Proof. By well-founded induction on $\langle O, U \rangle$ with respect to the lexicographic order of OBDD edges, we can assume that the theorem holds for all recursive calls.

We proceed by case analysis.

We give a detailed proof for the case in which $\text{root}(O)$ and $\text{root}(U)$ are nonterminals with the same variable index and $\langle \text{root}(O), \text{root}(U) \rangle \notin H$, which corresponds to lines *1, *2 and *3 in algorithm 2.

Let $\langle V^1, H^1 \rangle$ be the state after we call apply_helper recursively in line *1, and let r_1 be the resulting pointer from that call.

We assume that the Apply invariant holds for the initial state $\langle V, H \rangle$, thus by I.H. we have:

(I1) The Apply invariant holds for $\langle V^1, H^1 \rangle$.

(I2) $H^1[\text{low}(\text{root}(O)), \text{low}(\text{root}(U))] = r_1$.

(I3) $\text{size}(V) \leq \text{size}(V^1)$.

(I4) For all pointers o, u:

  (I4a) If $\langle o, u \rangle \notin H^1$, then also $\langle o, u \rangle \notin H$.
  (I4b) For all pointers w, if $H[o, u] = w$ then $H^1[o, u] = w$.
  (I4c) If $\langle o, u \rangle \notin H$ but $\langle o, u \rangle \in H^1$, then o is reachable from $\text{low}(\text{root}(O))$ and u is reachable from $\text{low}(\text{root}(U))$.

Let $\langle V^2, H^2 \rangle$ be the state after we call apply_helper recursively in line *2, and let r_2 be the resulting pointer from that call.

By assumption (I1), the Apply invariant holds for $\langle V^1, H^1 \rangle$. Thus we can apply our I.H. also to the recursive call in line *2, and obtain:

(I5) The Apply invariant holds for $\langle V^2, H^2 \rangle$.

(I6) $H^2[\text{high}(\text{root}(O)), \text{high}(\text{root}(U))] = r_2$.

(I7) $\mathrm{size}(V^1) \le \mathrm{size}(V^2)$.

(I8) For all pointers o, u:

- (I8a) If $\langle o, u\rangle \notin H^2$, then also $\langle o, u\rangle \notin H^1$.
- (I8b) For all pointers w, if $H^1[o, u] = w$ then $H^2[o, u] = w$.
- (I8c) If $\langle o, u\rangle \notin H^1$ but $\langle o, u\rangle \in H^2$, then o is reachable from $\mathrm{high}(\mathrm{root}(O))$ and u is reachable from $\mathrm{high}(\mathrm{root}(U))$.

First, we need to show that the Apply invariant holds for the final state $\langle V', H'\rangle$, where V' is V^2 with an additional node v such that $\mathrm{var}(v) = \mathrm{var}(O) = \mathrm{var}(U)$, $\mathrm{low}(v) = r_1$ and $\mathrm{high}(v) = r_2$, and H' is obtained from H^2 by setting $H^2[\mathrm{root}(O), \mathrm{root}(U)]$ to a pointer $r = \mathrm{size}(V_2)$ to the new node v at the end of V':

- To show part 1 of the Apply invariant, let V'_j be a node in V' at index j. We need to show that if $\mathrm{low}(V_j)$ is a nonterminal pointer to node V_k then $k < j$, and likewise for $\mathrm{high}(V_j)$.

  $j < \mathrm{size}(V') = \mathrm{size}(V^2) + 1$, thus $j \le \mathrm{size}(V^2)$. We distinguish between two cases: $j = \mathrm{size}(V^2)$ (j is the last index in V', in which case $V'_j = v$) or $j < \mathrm{size}(V^2)$:

$j = \mathrm{size}(V^2)$ Suppose $\mathrm{low}(V_j)$ points to node V_k. But $\mathrm{low}(V_j) = \mathrm{low}(v) = r_1$, so we get that r_1 points to node V_k at index k. By assumptions (I2) and (I8b), we get that $H^2[\mathrm{low}(\mathrm{root}(O)), \mathrm{low}(\mathrm{root}(U))] = r_1$. Since the invariant holds for $\langle V^2, H^2\rangle$ (assumption (I5)), we get that k is a valid pointer in V^2, so $k < \mathrm{size}(V^2) = j$, as needed.

Now suppose $\mathrm{high}(V_j) = \mathrm{high}(v) = r_2$ points to node $V_{k'}$. By (I5) and (I6) we get that k' is a valid pointer in V^2, so $k' < \mathrm{size}(V^2) = j$, as needed.

$j < \mathrm{size}(V^2)$ In case that $j < \mathrm{size}(V^2)$, we get $V'_j = V^2_j$. But by (I5), the invariant already holds for V^2, and in particular V^2_j has the needed property.

- To show part 2 of the Apply invariant, let o, u, w be pointers such that $H'[o, u] = w$.

  We again distinguish between two cases:

  - If $o = \mathrm{root}(O)$ and $u = \mathrm{root}(U)$, then we have $w = r$, and so w is a valid pointer that points to $V'_{\mathrm{size}(V^2)} = v$, and we have

    $$\mathrm{var}(v) = \mathrm{var}(O) = \mathrm{var}(U) = \min\{\mathrm{var}(O), \mathrm{var}(U)\} = \min\{\mathrm{var}(O[o]), \mathrm{var}(U[u])\}$$

    as needed to show part 2.a of the Apply invariant.

    We also have that $O[o] = O$ and $U[u] = U$ are ordered, by the assumption that O and U are OBDDs. This shows show parts 2.b and 2.c of the Apply invariant.

    For part 2.d, we need to show that $\langle V', r\rangle$ is also ordered. Since $\mathrm{low}(r) = r_1$ and $\mathrm{high}(r) = r_2$, it suffices to show that $\langle V', r_1\rangle$ and $\langle V', r_2\rangle$ are ordered, and that r may precede r_1 and r_2. By assumptions (I5) and (I6), we know that $\langle V^2, r_2\rangle$ is ordered. Hence $\langle V', r_2\rangle$ is also ordered because the additional node in V' does not affect the orderedness of $\langle V^2, r_2\rangle$ (in Lean, we formalize this statement in lemma `Apply.push_ordered`).

We also get that if r_2 points to a nonterminal V^2_j, then $\mathrm{var}(V^2_j) = \min\{\mathrm{var}(\mathrm{high}(O)), \mathrm{var}(\mathrm{high}(U))\}$, and thus $\mathrm{var}(V'_j) = \min\{\mathrm{var}(\mathrm{high}(O)), \mathrm{var}(\mathrm{high}(U))\}$. Since O and U are ordered, we have $\mathrm{var}(U) = \mathrm{var}(O) < \min\{\mathrm{var}(\mathrm{high}(O)), \mathrm{var}(\mathrm{high}(U))\}$, so r may precede r_2.

By similar reasoning, and using assumptions (I2), (I8b) and (I6), we get that $\langle V', r_1\rangle$ is ordered and that r may precede r_1. Thus $\langle V', r\rangle$ is ordered.

Lastly, we need to show that $D(\langle V', r\rangle) = D(O) \bullet D(U)$.

We have $D(\langle V', r_1\rangle) = D(\mathrm{low}(O)) \bullet D(\mathrm{low}(U))$ and $D(\langle V', r_2\rangle) = D(\mathrm{high}(O)) \bullet D(\mathrm{high}(U))$.

For brevity, let $i = \mathrm{var}(v) = \mathrm{var}(O) = \mathrm{var}(U)$. Thus:

$$\begin{aligned} D(\langle V', r\rangle) &= \boldsymbol{x}_i \cdot D(\langle V', r_2\rangle) + \bar{\boldsymbol{x}}_i \cdot D(\langle V', r_1\rangle) \\ &= \boldsymbol{x}_i \cdot (D(\mathrm{high}(O)) \bullet D(\mathrm{low}(U))) + \bar{\boldsymbol{x}}_i \cdot (D(\mathrm{high}(O)) \bullet D(\mathrm{low}(U))) \\ &= (\boldsymbol{x}_i \cdot D(\mathrm{high}(O)) + \bar{\boldsymbol{x}}_i \cdot D(\mathrm{low}(O))) \bullet (\boldsymbol{x}_i \cdot D(\mathrm{high}(U)) + \bar{\boldsymbol{x}}_i \cdot D(\mathrm{low}(U))) \\ &= D(O) \bullet D(U) \end{aligned}$$

To go from the second to the third line of the above equation, we rely on the fact that for all $c, b_1, b_2, b_3, b_4 \in \mathbb{B}$ (and for any $\bullet$) it holds that:

$$c \cdot (b_1 \bullet b_2) + \bar{c} \cdot (b_3 \bullet b_4) = (c \cdot b_1 + \bar{c} \cdot b_2) \bullet (c \cdot b_3 + \bar{c} \cdot b_4)$$

This corresponds to lemma `Apply.if_op3` in our formalization in Lean.

– Otherwise, if $\langle o, u\rangle \neq \langle \mathrm{root}(O), \mathrm{root}(U)\rangle$, then we have that $H^2[o, u] = w$.

Since the invariant holds for $\langle V^2, H^2\rangle$, we get:

9. If w is a nonterminal pointer to index j, then
   (a) $j < \mathrm{size}(V^2)$ and
   (b) $V^2_j = \min\{\mathrm{var}(O[o]), \mathrm{var}(U[u])\}$
10. $O[o]$ and $U[u]$ are ordered, and
11. $\langle V^2, w\rangle$ is ordered and $D(\langle V^2, w\rangle) = D(O[o]) \bullet D(U[u])$.

Thus we have $j < \mathrm{size}(V^2) < \mathrm{size}(V^2) + 1 < \mathrm{size}(V')$, and since $V'_j = V^2_j$ we also have $V'_j = \min\{\mathrm{var}(O[o]), \mathrm{var}(U[u])\}$.

It remains show that $\langle V', w\rangle$ is ordered and that $D(\langle V', w\rangle) = D(O[o]) \bullet D(U[u])$. These hold because of (I11) and the fact that V' is V^2 with an additional node (v) that is not reachable from w, and so it does not affect the orderedness and the denotation of $\langle V^2, w\rangle$.

We have thus established that the Apply invariant holds for the final state $\langle V', H'\rangle$.

We also need to show that $H'[\mathrm{root}(O), \mathrm{root}(U)] = r$, but that holds by construction of H'. Next, we need to show $\mathrm{size}(V) \leq \mathrm{size}(V')$. But we have $\mathrm{size}(V) \leq \mathrm{size}(V^1)$ (assumption (I3)) and $\mathrm{size}(V^1) \leq \mathrm{size}(V^2)$ ((I7)), thus $\mathrm{size}(V) \leq \mathrm{size}(V^2)$ holds by transitivity, and we get $\mathrm{size}(V) \leq \mathrm{size}(V')$ since $\mathrm{size}(V') = \mathrm{size}(V^2) + 1$.

Lastly, let o, u be pointers. We need to show:

- $\langle o, u\rangle \notin H'$ implies $\langle o, u\rangle \notin H$.
- For all pointers w, if $H[o, u] = w$ then $H'[o, u] = w$.

- If $\langle o, u\rangle \notin H$ but $\langle o, u\rangle \in H'$, then o is reachable from $\text{root}(O)$ and u is reachable from $\text{root}(U)$.

Suppose $\langle o, u\rangle \notin H'$. Then $\langle o, u\rangle \notin H^2$. But we have that $\langle o, u\rangle \notin H^2$ implies $\langle o, u\rangle \notin H^1$ and that $\langle o, u\rangle \notin H^1$ implies $\langle o, u\rangle \notin H$, so we get that $\langle o, u\rangle \notin H'$ implies $\langle o, u\rangle \notin H$.

Suppose that $H[o, u] = w$. We have that $H[o, u] = w$ implies $H^1[o, u] = w$ which implies $H^2[o, u] = w$, so we get $H^2[o, u] = w$. We need to show $H'[o, u] = w$, but H' and H^2 agree for all pairs of pointers except for $\langle \text{root}(O), \text{root}(U)\rangle$. Thus it now suffices to show that $\langle o, u\rangle \neq \langle \text{root}(O), \text{root}(U)\rangle$. By contradiction: assume $\langle o, u\rangle = \langle \text{root}(O), \text{root}(U)\rangle$. Thus $H^2[\text{root}(O), \text{root}(U)] = w$. We proceed by case distinction on $H^1[\text{root}(O), \text{root}(U)]$:

- If $H^1[\text{root}(O), \text{root}(U)] = w'$ for some w', then by assumption (I8b) we get that $H^2[\text{root}(O), \text{root}(U)] = w'$, and thus $w = w'$, so $H^1[\text{root}(O), \text{root}(U)] = w$. But we know that $\langle \text{root}(O), \text{root}(U)\rangle \notin H$, so we get from (I4c) that $\text{root}(O)$ is reachable from $\text{low}(\text{root}(O))$ and that $\text{root}(U)$ is reachable from $\text{low}(\text{root}(U))$. Contradiction, since O and U are ordered.

- Otherwise, if $\langle \text{root}(O), \text{root}(U)\rangle \notin H^1$, then from (I8c) we get that $\text{root}(O)$ is reachable from $\text{high}(\text{root}(O))$ and that $\text{root}(U)$ is reachable from $\text{high}(\text{root}(U))$. Contradiction.

This concludes the proof for the case in which $\text{root}(O)$ and $\text{root}(U)$ are nonterminals with the same variable index and $\langle \text{root}(O), \text{root}(U)\rangle \notin H$.

The other cases that involve recursive calls to apply_helper are similar. The two remaining cases—namely, the case in which $\langle \text{root}(O), \text{root}(U)\rangle \in H$ and the case in which both $\text{root}(O)$ and $\text{root}(U)$ are terminals—are simpler to verify. □

Corollary 43 (Correctness of algorithm 2)**.** *Given OBDDs O and U and operator $\cdot \bullet \cdot$, algorithm 2 outputs an ordered BDD W with $D(W) = D(O) \bullet D(U)$.*

Proof. The initial state of the algorithm satisfies the Apply invariant (definition 41) by virtue of V and H being empty. Hence by theorem 42, we have that the end state $\langle V', H'\rangle$ also satisfies the invariant, and we have $H'[\text{root}(O), \text{root}(U)] = \text{root}(W)$.

Thus by clause 2.d of the invariant, W is ordered and $D(W) = D(O) \bullet D(U)$. □

We implement algorithm 2 and formalize the proofs of theorem 42 and corollary 43 in the file `Apply.lean`. There are a few technical details about our formalization in Lean that deserve a detailed explanation.

First, note that although we described algorithm 2 as operating on two OBDDs with the same input size $O, U \in BDD_n$, the algorithm works just as well also if O and U have different input sizes. In general, for $O \in BDD_{n_O}$ and $U \in BDD_{n_U}$, algorithm 2 produces an OBDD W with input size $\max\{n_O, n_U\}$, since all nodes in W have variable indices that come from either O or U. We similarly generalize our characterization of the denotation of W from $D(W) = D(O) \bullet D(U)$ to $D(W) = D(\mathsf{lift}(O)) \bullet D(\mathsf{lift}(U))$.

Accordingly, in our implementation, the type of O is `OBdd n m` and the type of U is `OBdd n' m'`, where both `n` and `n'`, and `m` and `m'`, may differ.

Also note that when we invoke algorithm 2, we do not know in advance what will be the size of the heap of W. Since our type for (O)BDDs in Lean is indexed by the size of the heap, our `Apply.oapply` function needs to return the resulting heap size `s` as additional data along with the OBDD itself:

```
def oapply (op : Bool → Bool → Bool) (O : OBdd n m) (U : OBdd n' m') :
  (s : Nat) ×
  { OU : OBdd (n ⊔ n') s //
    ∀ I,
      OBdd.evaluate OU I =
      (op
        (OBdd.evaluate O (Vector.cast (by simp) (I.take n)))
        (OBdd.evaluate U (Vector.cast (by simp) (I.take n')))) }
```

Next, recall that in our concrete representation of BDDs as terms of type `Bdd n m`, the heap has type `Vector (Node n m) m`, which means that the type of individual nodes is indexed by `m`, the overall size of the vector which contains them. If we would push another node to the end of a `Vector (Node n m) m`, we would get a `Vector (Node n m) (m + 1)`, and if we would keep pushing more nodes we would get a `Vector (Node n m) (m + k)`, which means that all pointers in the `m + k` nodes of the vectors may only point to the first `m` indices. This is clearly not what we want, especially seeing as we start with an empty heap (`m = 0`),

Instead, we want to represent the under-construction heap V as an array of “raw” nodes, which are not indexed by the overall heap size, but contain all of the data that makes up a regular node otherwise. This in turn requires replacing the `Pointer m` type of safe pointers that we use for the low and high pointers of each node with an alternative `RawPointer` type of “raw” pointers, which are not indexed by the size of the vector into which they point:

```
private def RawPointer := Bool ⊕ Nat

private structure RawNode (n) where
  va : Fin n
  lo : RawPointer
  hi : RawPointer
```

Note that a `RawPointer` uses an unbounded `Nat` for vector indices, whereas `Pointer m` uses a `Fin m`. Thus a `RawPointer` is able to represent pointers to invalid (out-of-bounds) indices. To express that a given `RawPointer` is in fact valid with respect to vectors of a certain length `m`, we define a predicate `RawPointer.Bounded m`, which holds for raw pointers that are either terminal or that point to indices below `m`:

```
private def RawPointer.Bounded (m : Nat) (p : RawPointer) := ∀ {i}, p = .inr i → i < m
```

We then define a function that turns a `RawPointer` to a ready-to-use `Pointer m`, which we call `RawPointer.cook`. This function uses `RawPointer.Bounded` as a precondition:

```
private def RawPointer.cook (p : RawPointer) (h : p.Bounded m) : Pointer m
```

We lift the definitions of `Bounded` and `cook` from pointers to nodes, as follows:

```
private def RawNode.Bounded (m : Nat) (N : RawNode n) := N.lo.Bounded m ∧ N.hi.Bounded m

private def RawNode.cook (N : RawNode n) (h : N.Bounded m) : Node n m :=
  ⟨N.va, N.lo.cook h.1, N.hi.cook h.2⟩
```

Lastly we define a function that turns a vector of raw nodes to a proper heap, under the assumption that all pointers in the raw heap point from higher indices to lower indices, which guarantees that all pointers are valid, and corresponds to the first condition in the Apply invariant (definition 41):

```
private def cook_heap (v : Vector (RawNode n) m) :
  (∀ i : Fin m, v[i].Bounded i) → Vector (Node n m) m
```

We represent the state $\langle V, H\rangle$ in algorithm 2 as a structure `Apply.State`, which is defined as follows:

```
private structure State (n) (n') (m) (m') where
  size  : Nat
  heap  : Vector (RawNode (max n n')) size
  cache : Std.HashMap (Pointer m × Pointer m') RawPointer
```

The `State.heap` field corresponds to the array of nodes V, and `State.cache` corresponds to the hash table H.

Note that we also have an additional field, `State.size`, which is just the size of `State.heap`, corresponding to size(V). This may seem redundant at first, since a `Vector` α `n` is just an `Array` α of size `n`, so we could just as well have defined `State.heap` to be an `Array (RawNode (max n n'))`, and use `State.heap.size` whenever we would want to use `State.size`, as in the following alternative definition:

```
private structure State' (n) (n') (m) (m') where
  heap  : Array (RawNode (max n n'))
  cache : Std.HashMap (Pointer m × Pointer m') RawPointer
```

However, the separate `size` field in `State` leads to a significant performance improvement due to a somewhat subtle reason.

In Lean, pushing a node to the end of an array is very efficient ($O(1)$ time and space, amortized) if, and only if, Lean can determine that the original array is never used afterwards. Otherwise, Lean has to make a copy of the original array and extend the copy instead, to keep the original array intact, which requires linear time and space in the size of the array. This decision occurs in the function `lean_array_push` which is part of Lean's low-level `Array` implementation.

Now, in the push_node subroutine of algorithm 2, we push a node to the end of `State.heap`, and then we return the index of the new node, which is the size that `State.heap` had *before* we pushed the new node. By keeping the size of `State.heap` as a separate piece of data `State.size`, we avoid referring to `State.heap.size` after extending `State.heap` in our implementation of push_node, which ensures that Lean can determine that `State.heap` has no other references, and avoid costly copies of the underlying array. Other techniques, such as storing `State.heap.size` in a variable before extending `State.heap` and similar semantics-preserving code transformations, have proven ineffective in our tests, which used Lean version 4.20.

We formalize the Apply invariant from definition 41 in the predicate `Apply.Invariant`:

```
private def Invariant
    (op : Bool → Bool → Bool) (O : OBdd n m) (U : OBdd n' m') (s : State n n' m m') :=
  ∃ hh : (∀ i : Fin s.size, RawNode.Bounded i s.heap[i]),    -- part 1
  ∀ (k : (Pointer m × Pointer m')) (p : RawPointer),         -- part 2
    s.cache[k]? = some p →
    (∀ j h, p = .inr j →
      (s.heap[j]'h).va.1 = (toVar O.1.heap k.1) ⊓ (toVar U.1.heap k.2)) ∧
    ∃ (hk1 : Bdd.Ordered ⟨O.1.heap, k.1⟩) (hk2 : Bdd.Ordered ⟨U.1.heap, k.2⟩),
      ∃ hp : p.Bounded s.size,
        ∃ o : Bdd.Ordered ⟨cook_heap s.heap hh, p.cook hp⟩,
          ∀ I,
            OBdd.evaluate ⟨⟨cook_heap s.heap hh, p.cook hp⟩, o⟩ I =
            (op
              (OBdd.evaluate ⟨⟨O.1.heap, k.1⟩, hk1⟩ (Vector.cast (by simp) (I.take n)))
              (OBdd.evaluate ⟨⟨U.1.heap, k.2⟩, hk2⟩ (Vector.cast (by simp) (I.take n'))))
```

Next, we define the initial state of algorithm, and prove that the invariant holds for it, which follows directly from the fact that the initial hash table and vector of nodes are both empty:

```
private def initial : State n n' m m' :=
  ⟨0, Vector.emptyWithCapacity 0, Std.HashMap.emptyWithCapacity 0⟩

private lemma inv_initial {op} {O} {U} : Invariant op O U initial
```

Finally, we present the type of `Apply.apply_helper`, which implements the apply_helper function from algorithm 2, along with its correctness proof:

```
private def apply_helper
    (op : Bool → Bool → Bool) (O : OBdd n m) (U : OBdd n' m')
    (s : State n n' m m') (inv : Invariant op O U s) :
  { r : State n n' m m' × RawPointer //
    Invariant op O U r.1 ∧
    (r.1.cache[(⟨O.1.root, U.1.root⟩)]? = some r.2) ∧
    (s.size ≤ r.1.size) ∧
    (∀ (k : Pointer m × Pointer m'),
      (r.1.cache[k]? = none → s.cache[k]? = none) ∧
      (∀ p, s.cache[k]? = some p → r.1.cache[k]? = some p) ∧
      (s.cache[k]? = none → (∃ p, r.1.cache[k]? = some p) →
        Reachable O.1.heap O.1.root k.1 ∧ Reachable U.1.heap U.1.root k.2))
  }
```

The implementation of `Apply.apply_helper` closely follows the pseudo-code of algorithm 2, except that it also constructs and returns a proof that its output has the correct properties, along the lines of the proof that we saw above for theorem 42.

In [Bry86], Bryant points at one additional optimization that we have not yet included in our description, and implementation, of algorithm 2: given more information about the binary Boolean operator $\bullet$, it is possible to eliminate more recursive calls to apply_helper. Namely, for some values of $\bullet$, we can determine that the $D(O) \bullet D(U)$ is a constant function, and return the appropriate terminal without additional recursive calls, even if $\mathrm{root}(O)$ and $\mathrm{root}(U)$ are not both terminals. For example, if $\bullet$ is $+$ (disjunction) and $\mathrm{root}(O)$ is a terminal with value true, then we can already tell that $D(O) \bullet D(U) = \mathbf{1}$, even if $\mathrm{root}(U)$ is nonterminal.

We left this optimization out of the current implementation to simplify the proof of correctness. In section 3.1, we outline a path towards enhancing our implementation with this optimization.

Restrict

The `BDD.restrict` function is similar in many ways to `BDD.apply`: both functions construct an ROBDD whose denotation is a certain function of the denotations of existing ROBDDs, which `apply` and `restrict` take as inputs. In other words, both functions implement operations on Boolean functions at the level of the ROBDDs that represent them—`apply` implements $\langle f, g\rangle \mapsto f \bullet g$ for Boolean functions

f, g and a binary Boolean operator $\bullet$, while `restrict` implements $f \mapsto f_{i \leftarrow b}$ for some Boolean b and variable index i.

The implementations of `restrict` is also similar to the implementation of `apply`: the core of `BDD.restrict` is the function `Restrict.orestrict`, which traverses an OBDD O (not necessarily reduced) in depth-first order, starting from its root, and produces a new OBDD W such that $D(W) = D(O)_{i \leftarrow b}$. The key to OBDD restriction is the observation that for all Boolean functions f, g, c we have:

$$(c \cdot f + \bar{c} \cdot g)_{i \leftarrow b} = c_{i \leftarrow b} \cdot f_{i \leftarrow b} + \bar{c}_{i \leftarrow b} \cdot g_{i \leftarrow b}$$

In Lean, we formalize this fact in lemma `Nary.restrict_if`:

```
lemma restrict_if {c : Func n α Bool} :
  restrict (fun I ↦ if c I then f I else g I) b i I =
  if (restrict c b i I) then (restrict f b i I) else (restrict g b i I) := rfl
```

Therefore, for an OBDD O with a nonterminal root, whose denotation is

$$D(O) = \boldsymbol{x}_{\mathrm{var}(O)} \cdot D(\mathrm{high}(O)) + \bar{\boldsymbol{x}}_{\mathrm{var}(O)} \cdot D(\mathrm{low}(O))$$

we can construct an OBDD with denotation $D(O)_{i \leftarrow b}$ by first recursively constructing OBDDs with denotations $D(\mathrm{high}(O))_{i \leftarrow b}$ and $D(\mathrm{low}(O))_{i \leftarrow b}$.

The pseudo-code for our restriction algorithm, which employs this recursive scheme, is given in algorithm 3.

To prove the correctness of algorithm 3, we proceed as we did in the case of algorithm 2. Namely, we first identify an invariant that the recursive helper function restrict_helper respects:

Definition 44 (Restrict Invariant). *Given an OBDD $O \in BDD_n$, a Boolean b, an index $i \in [n]$, a hash table H and an array of BDD nodes V,* Restrict invariant *is the conjunction of the following conditions:*

1. *For all nodes V_j in V, if $\mathrm{low}(V_j)$ is a nonterminal pointer to node V_k, then $k < j$; and likewise for $\mathrm{high}(V_k)$.*
2. *For all pointers o, w such that $H[o] = w$ we have:*
   (a) *If w is a nonterminal pointer, then:*
      i. *w points to a node V_j at a valid index $j < \mathrm{size}(V)$, and*
      ii. *if $\mathrm{var}(O[o]) = i$ then $\mathrm{var}(V_j) > \mathrm{var}(O[o])$, otherwise $\mathrm{var}(V_j) = \mathrm{var}(O[o])$*
   (b) *$O[o]$ is ordered.*
   (c) *The BDD $W = \langle V, w \rangle$ with heap V and root w is ordered, and $D(W) = D(O[o])_{i \leftarrow b}$.*

Next, we prove that restrict_helper preserves the invariant, and satisfies additional *post-conditions*:

Theorem 45 (Correctness of restrict_helper). *If the Restrict invariant holds for a given state $\langle V, H \rangle$ with respects to OBDD O, Boolean b and index i, then after invoking* restrict_helper(O) *and obtaining an output pointer r and a new state $\langle V', H' \rangle$, the following statements hold:*

1. *The invariant holds also for the new state $\langle V', H' \rangle$.*

```
**Input:** OBDD $O \in BDD_n$, $b \in \mathbb{B}$, $i \in [n]$
**Output:** OBDD $W$ with $D(W) = D(O)_{i \leftarrow b}$
$V \leftarrow$ empty array of BDD nodes
$H \leftarrow$ empty hash table mapping input pointers to output pointers
def restrict_helper($O$):
    if $H[\mathrm{root}(O)] = p$ for some pointer $p$ then
        return $p$
    else
        if $\mathrm{root}(O)$ is terminal then
            $H[\mathrm{root}(O)] \leftarrow \mathrm{root}(O)$
            return $\mathrm{root}(O)$
        else
            if $\mathrm{var}(O) = i$ then
                if $b = \mathsf{true}$ then
                    $h \leftarrow$ restrict_helper($\mathrm{high}(O)$)
                    $H[\mathrm{root}(O)] \leftarrow h$
                    return $h$
                else
                    $l \leftarrow$ restrict_helper($\mathrm{low}(O)$)
                    $H[\mathrm{root}(O)] \leftarrow l$
                    return $l$
                end
            else
                $l \leftarrow$ restrict_helper($\mathrm{low}(O)$)
                $h \leftarrow$ restrict_helper($\mathrm{high}(O)$)
                $s \leftarrow \mathrm{size}(V)$
                Push a new node $V_s$ at the end of $V$
                $\mathrm{var}(V_s) \leftarrow (\mathrm{var}(O))$, $\mathrm{low}(V_s) \leftarrow l$, $\mathrm{high}(V_s) \leftarrow h$
                $H[\mathrm{root}(O)] \leftarrow s$
                return $s$;
            end
        end
    end
$r \leftarrow$ restrict_helper($O$)$\rangle$
return BDD $\langle V, r \rangle$ with heap $V$ and root $r$
```

Algorithm 3: Restrict

2. $H'[\mathrm{root}(O)] = r$.

3. $\mathrm{size}(V) \leq \mathrm{size}(V')$.

4. *For all pointers o:*

   (a) *$o \notin H'$ implies $o \notin H$.*

   (b) *For all pointers w, if $H[o] = w$ then $H'[o] = w$.*

   (c) *If $o \notin H$ but $o \in H'$, then o is reachable from* $\mathrm{root}(O)$.

Proof. By induction on O, we can assume that the theorem holds for all recursive calls. We proceed by case analysis, analogously to the proof of theorem 42. □

We implement algorithm 3 and formalize its correctness proof in the Lean functions `Restrict.orestrict` and `Restrict.restrict_helper`:

```
private def restrict_helper (O : OBdd n m) (b : Bool) (i : Fin n)
    (s : State n m) (inv : Invariant b i O s) :
  { r : State n m × RawPointer //
    (Invariant b i O r.1)             ∧
    (r.1.cache[O.1.root]? = some r.2) ∧
    (s.size ≤ r.1.size)               ∧
    (∀ (k : Pointer m),
      (∀ p, s.cache[k]? = some p → r.1.cache[k]? = some p) ∧
      (r.1.cache[k]? = none → s.cache[k]? = none)          ∧
      (s.cache[k]? = none → (∃ p, r.1.cache[k]? = some p) →
        Reachable O.1.heap O.1.root k))
  }

def orestrict (b : Bool) (i : Fin n) (O : OBdd n m) :
  (s : Nat) × { W : OBdd n s // W.evaluate = Nary.restrict O.evaluate b i }
```

Our formulation of BDD restriction in algorithm 3 is conducive for proving correctness in the formal setting of Lean. As Bryant remarks in [Bry86], we can also construct the restriction of an OBDD O at some index i just by modifying all pointers reachable from $\text{root}(O)$ that point to nodes with variable index i—thus changing the input BDD O—instead of constructing a completely new BDD as we do algorithm 3. We describe this alternative restriction algorithm in pseudo-code in algorithm 4.

Note that in algorithm 4, the recursive helper function has just one recursive case, which is the case wherein $\text{root}(O)$ is not in the cache H and it is a nonterminal with variable index $\text{var}(O) < i$. When $\text{var}(O) \geq i$, algorithm 4 does not need to descend further down O, since the orderedness of O guarantees that no pointers to nodes with variable index i are reachable from $\text{root}(O)$. Thus if all relevant nodes with variable index i are close to $\text{root}(O)$ (and far from the terminal nodes), then algorithm 4 may require significantly less recursive calls than algorithm 3, which always descends all the way down to the terminal nodes. In general, both algorithm 3 and algorithm 4 have $O(\text{size}(O))$ time complexity.

Reduce

Both `Apply.oapply` and `Restrict.orestrict` produce OBDDs with a given target denotation. In general, neither of these algorithms produce *reduced* OBDDs, however. To obtain ROBDDs, we employ an additional *reduction* algorithm, which reduces an arbitrary OBDD O to an ROBDD R with the same denotation $D(R) = D(O)$.

Our implementation closely follows the *Reduce* algorithm from [Bry86]. The corresponding function in `Reduce.lean` is `Reduce.oreduce`:

```
def oreduce (O : OBdd n m) : (s : Nat) × OBdd n s
```

We saw in theorem 31 that ROBDDs are the smallest OBDDs with a given denotation. In particular, when we reduce an OBDD O to an ROBDD R, we have $\text{size}(R) < \text{size}(O)$, except when O is already reduced, in which case we get $\text{size}(R) = \text{size}(O)$ of course. The `Reduce.oreduce` function thus produces an `OBdd` with a (potentially) different heap size than the heap size of its input, so it also returns the new heap size `s` alongside the resulting ROBDD, similarly to `Apply.oapply`.

Formally proving the correctness of `Reduce.oreduce` in Lean is very involved. Since our implementation closely follows the well-known *Reduce* algorithm from [Bry86], we opt to leave this correctness

```
Input: OBDD $O \in BDD_n$, $b \in \mathbb{B}$, $i \in [n]$
Output: OBDD $W$ with $D(W) = D(O)_{i \leftarrow b}$
$H \leftarrow$ empty hash table mapping input pointers to output pointers
def restrict_alt_helper($O$):
    if $H[\text{root}(O)] = p$ for some pointer $p$ then
        return $p$
    else
        if $\text{root}(O)$ is terminal or $\text{var}(O) > i$ then
            return $\text{root}(O)$
        else
            if $\text{var}(O) = i$ then
                if $b = \text{true}$ then
                    $H[\text{root}(O)] \leftarrow \text{high}(\text{root}(O))$
                    return $\text{high}(\text{root}(O))$
                else
                    $H[\text{root}(O)] \leftarrow \text{low}(\text{root}(O))$
                    return $\text{low}(\text{root}(O))$
                end
            else
                $l \leftarrow$ restrict_alt_helper($\text{low}(O)$)
                $h \leftarrow$ restrict_alt_helper($\text{high}(O)$)
                $\text{low}(\text{root}(O)) \leftarrow l$
                $\text{high}(\text{root}(O)) \leftarrow h$
                $H[\text{root}(O)] \leftarrow \text{root}(O)$
                return $\text{root}(O)$;
            end
        end
    end
$r \leftarrow$ restrict_alt_helper($O$)$\rangle$
return $O[r]$
```

Algorithm 4: RestrictAlt

proof for future work. We state the correctness lemmas for `Reduce.oreduce` and take them as given for now:

```
lemma oreduce_reduced {O : OBdd n m} : OBdd.Reduced (oreduce O).2 := sorry

lemma oreduce_evaluate {O : OBdd n m} : (oreduce O).2.evaluate = O.evaluate := sorry
```

The first step of the *Reduce* algorithm is to collect all nonterminal nodes reachable from the root of the OBDD that we want to reduce, and put them in different "buckets" labeled by their variable indices. In Bryant's description of the algorithm in [Bry86], this is stated as "[p]ut each vertex u on list vlist[u.index]." To do so, we implement a function `Collect.collect`, which collects all relevant nonterminal nodes for a given OBDD into a list. It uses a recursive helper function `collect_helper` to traverse the OBDD in depth-first order, accumulating newly encountered node indices:

```
private def collect_helper (O : OBdd n m) :
    Vector Bool m × List (Fin m) → Vector Bool m × List (Fin m) :=
  match h : O.1.root with
  | .terminal _ => id
  | .node j     => fun s ↦
```

```
    if s.1.get j
    then s
    else collect_helper (O.high h) (collect_helper (O.low h) ⟨s.1.set j true, j :: I.2⟩)
termination_by O

def collect (O : OBdd n m) : List (Fin m) :=
  (collect_helper O ⟨Vector.replicate m false, []⟩).2
```

We formally prove the correctness of `Collect.collect` in the following two lemmas, which state that the list of the node indices that the function produces contains each relevant nonterminal, without duplications and with no other unrelated elements:

```
lemma mem_collect_iff_reachable {O : OBdd n m} {j : Fin m} :
  j ∈ collect O ↔ Reachable O.1.heap O.1.root (.node j)

lemma collect_nodup {O : OBdd n m} : (collect O).Nodup
```

We use these lemmas to prove the `Size.size_spec` lemma, stating that the length of `collect O` is exactly the size of O, which we saw in section 2.2.2.

Relabel

The last primitive our library provides for transforming BDDs, in addition to `BDD.apply` and `BDD.restrict`, is the `BDD.relabel` function, which we have described in section 2.1.3. This interface function is implemented on top of an internal function called `Relabel.relabel`, which modifies all variable indices in the heap of a given BDD $B \in BDD_n$ according to a given function $f : \mathbb{N} \to \mathbb{N}$:

```
def relabel_node {f : Nat → Nat} (hf : ∀ i : Fin n, f i < f n) : Node n m → Node (f n) m
  | ⟨var, low, high⟩ => ⟨⟨f var.1, hf _⟩, low, high⟩

def relabel_heap {f : Nat → Nat} (hf : ∀ i : Fin n, f i < f n) :
    Vector (Node n m) m → Vector (Node (f n) m) m := Vector.map (relabel_node hf)

def relabel {f : Nat → Nat} (hf : ∀ i : Fin n, f i < f n) : Bdd n m → Bdd (f n) m
  | ⟨heap, root⟩ => ⟨relabel_heap hf heap, root⟩
```

Note that unlike `BDD.relabel`, the lower-level `Relabel.relabel` operates on arbitrary BDDs, which need not be reduced or even ordered. Its only precondition is that the function f, which relabels individual variable indices, maps all variables indices $i \in [n]$ to new variable indices below $f(n)$, which is the input size of the resulting BDD. This is more of a convention than a condition, though: since n is not itself a valid variable index ($n \notin [n]$), callers of `Relabel.relabel` can have f map n to any number at all. Thus $f(n)$ simply specifies the input size of the resulting BDD.

Next, we prove that `Relabel.relabel` produces ordered BDDs, under two condition:

1. The input BDD $B \in BDD_n$ is ordered.

2. f is strictly increasing for variable indices that B uses, that is variable indices $i \in [n]$ such that some nonterminal node v reachable from $\text{root}(B)$ has $\text{var}(v) = i$.

```
def usesVar (B : Bdd n m) (i : Fin n) :=
  ∃ j, Reachable B.heap B.root (node j) ∧ B.heap[j].var = i

lemma relabel_ordered {B : Bdd n m} {f : Nat → Nat} {hf : ∀ i : Fin n, f i < f n} :
  (∀ i i' : Fin n, i < i' → B.usesVar i → B.usesVar i' → f i < f i') →
  Bdd.Ordered B → Bdd.Ordered (relabel hf B)
```

We then use the `relabel_ordered` lemma to lift the `Relabel.relabel` to a function `Relabel.orelabel`, which takes and produces OBDDs, rather arbitrary BDDs:

```
def orelabel (O : OBdd n m) {f : Nat → Nat} (hf : ∀ i : Fin n, f i < f n)
    (hu : ∀ i i' : Fin n, i < i' → O.1.usesVar i → O.1.usesVar i' → f i < f i') :
  OBdd (f n) m := ⟨(relabel hf O.1), relabel_ordered hu O.2⟩
```

Recall that the purpose of BDD relabeling is to change the set of variable indices that the BDD's denotation depend on. More concretely, if we relabel an OBDD $O \in BDD_n$ with a relabeling function f, the resulting OBDD W should satisfy $D(W)(x_0, \ldots, x_{f(n)-1}) = D(O)(x_{f(0)}, x_{f(1)}, \ldots, x_{f(n-1)})$.

To establish the correctness of our `orelabel` function, we prove the following lemma:

```
theorem orelabel_evaluate (O : OBdd n m) {f : Nat → Nat} {hf : ∀ i : Fin n, f i < f n}
    {hu : ∀ i i' : Fin n, i < i' → O.1.usesVar i → O.1.usesVar i' → f i < f i'}
    {I : Vector Bool (f n)} :
  OBdd.evaluate (orelabel O hf hu) I = O.evaluate (Vector.ofFn (fun i ↦ I[f i]'(hf i)))
```

Lastly we prove that `orelabel` also preserves the property of being reduced, so it takes ROBDDs to ROBDDs:

```
lemma orelabel_reduced {O : OBdd n m} {f : Nat → Nat} {hf : ∀ i : Fin n, f i < f n}
    {hu : ∀ i i' : Fin n, i < i' → O.1.usesVar i → O.1.usesVar i' → f i < f i'} :
  O.Reduced → (orelabel O hf hu).Reduced
```

Recall that the interface function `BDD.relabel`, unlike the underlying `Relabel.orelabel`, does not mention the `Bdd.usesVar` predicate in its precondition or otherwise, which is crucial since our library's interface deals only with the Boolean functions that BDDs denote, never exposing the specifics of how BDDs represent these Boolean functions. Thus users of our library cannot (and should not) reason about which variables a BDD uses in the sense of `Bdd.usesVar`.

Instead, we must provide a purely semantic characterization of the variable indices that a given BDD uses, which users can reason about exclusively from the point of view of the BDD's denotation. Here, the tight connection between the structure of an ROBDD and its denotation comes to our aid, as captured in the following theorem, which allows us to express statements about the variables an ROBDD uses in terms of dependencies of the Boolean function it denotes:

Theorem 46. *For an ROBDD $O \in BDD_n$, O uses a variable index $i \in [n]$ if and only if $D(O)$ depends on i.*

Proof. Let $O \in BDD_n$ be an ROBDD.

⇒ Suppose that O uses a variable index $i \in [n]$. Thus there exists a nonterminal node v such that $\mathrm{var}(v) = i$ and v is reachable from $\mathrm{root}(O)$, which means that there is a path P from $\mathrm{root}(O)$ to a pointer to v.

By induction on the length of P:

Base If P is of length 0, we get that $\mathrm{root}(O)$ points to v directly. Hence we have:

$$D(O) = \boldsymbol{x}_i \cdot D(\mathrm{high}(O)) + \bar{\boldsymbol{x}}_i \cdot D(\mathrm{low}(O))$$

Thus $D(\mathrm{high}(O)) = D(O)_{i \leftarrow \mathsf{true}}$ and $D(\mathrm{low}(O)) = D(O)_{i \leftarrow \mathsf{false}}$. Suppose that $D(O)$ does not depend on i, thus $D(O)_{i \leftarrow \mathsf{true}} = D(O)_{i \leftarrow \mathsf{false}}$, and hence $D(\mathrm{high}(O)) = D(\mathrm{low}(O))$. However, $\mathrm{high}(O)$ and $\mathrm{low}(O)$ are reduced, since they are reachable from the ROBDD O. By canonicity (theorem 25), we get that $\mathrm{high}(O) \sim \mathrm{low}(O)$. But that means that O is not reduced. Contradiction. Thus $D(O)$ depends on i, as needed.

Step If P is of length $\ell + 1$, then $\mathrm{root}(O)$ is a nonterminal and v is reachable from either $\mathrm{low}(O)$ or $\mathrm{high}(O)$ via a path of ℓ edges. If v is reachable from $\mathrm{low}(O)$ via a path of ℓ edges, then in particular that means that $\mathrm{low}(O)$ uses i. By I.H. we now get that $D(\mathrm{low}(O))$ depends on i. Thus there exists a vector $x \in \mathbb{B}^n$ with $x_i = \mathsf{false}$ such that $D(\mathrm{low}(O))(x) \neq D(\mathrm{low}(O))(x_0, \dots, x_{i-1}, \mathsf{true}, x_{i+1}, \dots, x_{n-1})$.

Let $x' = x_0, \dots, x_{\mathrm{var}(O)-1}, \mathsf{false}, x_{\mathrm{var}(O)+1}, \dots, x_{n-1}$ be the vector that is obtained from x by setting the variable at index $\mathrm{var}(O)$ to false. Since O is ordered, $\mathrm{low}(O)$ does not depend on $\mathrm{var}(O)$, and thus $\mathrm{low}(O)(x') = \mathrm{low}(O)(x)$. Hence we have:

$$\begin{aligned}
D(O)(x') &= D(\mathrm{low}(O))(x') \\
&= D(\mathrm{low}(O))(x) \\
&\neq D(\mathrm{low}(O))(x_0, \dots, x_{i-1}, \mathsf{true}, x_{i+1}, \dots, x_{n-1}) \\
&= D(\mathrm{low}(O))(x_0, \dots, x_{\mathrm{var}(O)-1}, \mathsf{false}, x_{\mathrm{var}(O)+1}, \dots, x_{i-1}, \mathsf{true}, x_{i+1}, \dots, x_{n-1}) \\
&= D(\mathrm{low}(O))(x'_0, \dots, x'_{i-1}, \mathsf{true}, x'_{i+1}, \dots, x'_{n-1}) \\
&= D(O)(x'_0, \dots, x'_{i-1}, \mathsf{true}, x'_{i+1}, \dots, x'_{n-1})
\end{aligned}$$

Thus O depends on i, as needed. We handle the case in which v is reachable from $\mathrm{high}(O)$ rather than $\mathrm{low}(O)$ analogously.

$\Leftarrow$ By contraposition, suppose that O does not use i. We show that $D(O)$ does not on i by well-founded induction on O. If $\mathrm{root}(O)$ is a terminal, then $D(O)$ is constant and in particular it does not depend on i, as needed. Otherwise, if $\mathrm{root}(O)$ is nonterminal, then since O does not use i we know that $\mathrm{var}(O) \neq i$ and that $\mathrm{low}(O)$ and $\mathrm{high}(O)$ do not use i either. Thus by I.H. we get that $D(\mathrm{low}(O))$ and $D(\mathrm{high}(O))$ do not depend on i, and by definition of BDD denotation we conclude that $D(O)$ does not depend i. □

We formalize theorem 46 and its proof in the following Lean lemma:

```
lemma OBdd.usesVar_iff_dependsOn_of_reduced {O : OBdd n m} :
  O.Reduced → (O.1.usesVar i ↔ Nary.DependsOn O.evaluate i)
```

With this lemma in place, we are able to translate the precondition of `BDD.relabel`, which is stated in terms of variables that an ROBDD's denotation depend on, to the precondition of the underlying `Relabel.orelabel`, which is stated in terms of variables that the BDD uses, as we saw above.

Another difference between the interface function `BDD.relabel` and the lower-level `Relabel.orelabel` is the type of the relabeling function f that they accept. With `BDD.relabel`, f has type `Fin B.nvars → Fin n` where `B.nvars` is the input size of the input BDD, and `n` is the input size of the *output* BDD, which callers can specify explicitly. This stands in contrast with the `Nat → Nat` type of the relabeling function that the underlying `Relabel.orelabel` accepts, as we saw above. The reason for this difference is that we expect the interface that `BDD.relabel` provides, with the ability to explicitly specify the resulting input size, to be more convenient for users of our library. Under the hood, `BDD.relabel` converts the `Fin B.nvars → Fin n` relabeling function to an appropriate `Nat → Nat` function before passing it to `Relabel.orelabel`.

2.2.6 Deciding Variable Dependencies

Theorem 46, in addition to its utility for the definition of `BDD.relabel`, also induces an efficient algorithm for deciding whether the denotation of an ROBDD depends on a given variable index. For an arbitrary n-ary Boolean function f, deciding whether f depends on index $i \in [n]$ requires checking potentially all 2^n possible inputs. However, if we have an ROBDD B such that $D(B) = f$, then we can check whether f depends on i in $O(\text{size}(B))$, by deciding whether B *uses* i, and leverage theorem 46 to translate this decision to a decision about to the dependency of f on i.

We show a decision procedure for whether a given OBDD uses a given variable index in algorithm 5.

```
Input: OBDD $O \in BDD_n$, $i \in [n]$
Output: true if $O$ uses $i$, false otherwise
$F \leftarrow \emptyset$
def uv_helper($O$):
    if $\text{root}(O)$ is terminal or $\text{var}(O) > i$ or $\text{root}(O) \in F$ then
        return false
    else
        if $\text{var}(O) = i$ then
            return true
        else
            if uv_helper($\text{low}(O)$) then
                return true
            else
                if uv_helper($\text{high}(O)$) then
                    return true
                else
                    $F \leftarrow F \cup \{\text{root}(O)\}$
                    return false
                end
            end
        end
    end
return uv_helper($O$)
```

Algorithm 5: ROBDD variable usage decision procedure

We formalize algorithm 5 in Lean with the following `DecidablePred` instance:

```
instance OBdd.instDecidableUsesVar {O : OBdd n m} : DecidablePred O.1.usesVar
```

Note that algorithm 5 and its implementation in `instDecidableUsesVar` do not require that the input OBDD O is reduced. However, to apply theorem 46 and conclude that $D(O)$ depends (or rather does not depend) on a given index, we must know that O is reduced.

With instDecidableUsesVar and usesVar_iff_dependsOn_of_reduced (which implement algorithm 5 and theorem 46, respectively), we implement the DecidablePred instance instDecidableDependsOn, which we saw in section 2.1.4 as part of our description of the library's interface.

```
instance instDecidableDependsOn (B : BDD) : DecidablePred (DependsOn B.denotation') :=
  fun i ↦
    (show B.denotation' = B.obdd.evaluate by simp [...]) ▸
    (decidable_of_iff _ (OBdd.usesVar_iff_dependsOn_of_reduced B.hred))
```

2.2.7 Implementation of Choice and Find

Lastly, we describe the implementation the functions that our library provides for finding "solutions"—inputs for which the denotation of a given BDD return true. These functions are BDD.choice and BDD.find, which we saw in section 2.1.5.

The BDD.choice function is implemented on top of an internal function Choice.choice, which takes an ROBDD $O \in BDD_n$ and returns an input vector $x \in \mathbb{B}^n$ such that $D(O)(x) = \mathsf{true}$, under the precondition that such an input exists. In fact, it returns the lexicographically smallest x with that property, using an algorithm that Knuth briefly mentions in [Knu09]. The pseudo-code for this algorithm is given in algorithm 6.

```
Input: ROBDD $O \in BDD_n$ for which there exists $x \in \mathbb{B}^n$ such that $D(O)(x) = \mathsf{true}$
Output: $x \in \mathbb{B}^n$ such that $D(O)(x) = \mathsf{true}$
def choice_helper($O \in BDD_n$ with nonterminal $\mathrm{root}(O)$, $x \in \mathbb{B}^n$):
    if $\mathrm{low}(\mathrm{root}(O))$ is nonterminal then
        return choice_helper($\mathrm{low}(O)$, $x$)
    else
        if $\mathrm{val}(\mathrm{root}(O)) = \mathsf{true}$ then
            $x_{\mathrm{var}(O)} \leftarrow \mathsf{false}$ return $x$
        else
            if $\mathrm{high}(\mathrm{root}(O))$ is nonterminal then
                $x \leftarrow$ choice_helper($\mathrm{high}(O), x$)
            end
            $x_{\mathrm{var}(O)} \leftarrow \mathsf{true}$ return $x$
        end
    end
$x \leftarrow \langle \mathsf{false}, \ldots, \mathsf{false} \rangle$
if $\mathrm{root}(O)$ is terminal then   /* Must have $\mathrm{val}(\mathrm{root}(O)) = \mathsf{true}$ since $D(O)$ is not $\mathbf{0}$ */
    return $x$
else
    return choice_helper($O$, $x$)
end
```

Algorithm 6: ROBDD Choice

In [Bry86], Bryant presents an algorithm called *SatisfyOne*, which is similar to algorithm 6 except that Bryant's algorithm works also for OBDDs that are not reduced. It also starts with an arbitrary vector $x \in \mathbb{B}^n$ whereas we start with $x = \langle \mathsf{false}, \ldots, \mathsf{false}$ to obtain the lexicographically smallest solution. As Bryant notes, for ROBDDs, the *SatisfyOne* algorithm takes at most n recursive calls for an input ROBDD $O \in BDD_n$, since it traverses one path from $\mathrm{root}(O)$ to the true terminal, and each of the n variable indices may appear at most once along such a path. The same holds for our algorithm 6—its time complexity is linear in n. For non-reduced OBDDs, Bryant's *SatisfyOne* algorithm does not retain this performance guarantee: it may require up to 2^n recursive calls in the worst case.

We implement algorithm 6 in the function `Choice.choice`, and prove its correctness in the lemma `Choice.choice_evaluate`:

```
def choice (O : OBdd n m) : (∃ I, O.evaluate I) → Vector Bool n
def choice_evaluate {O : OBdd n m} (hr : O.Reduced) (ht : ∃ I, O.evaluate I = true) :
  O.evaluate (choice O ht) = true
```

The interface function `BDD.choice` is implemented as a wrapper around `Choice.choice`. The `BDD.find` function is in turn implemented on top of `BDD.choice`—it first checks whether its input denotes a satisfiable Boolean function by comparing it to `BDD.const false`, and if so it delegates to the `BDD.choice` function; otherwise it returns `none` to indicate that no solution exists:

```
private lemma find_aux {B : BDD} :
  ¬ B.SemanticEquiv (const false) → ∃ (I : Vector Bool B.nvars), B.denotation' I = true

def find {B : BDD} : Option (Vector Bool B.nvars) :=
  if h : B.SemanticEquiv (const false) then none else some (choice (find_aux h))
```

2.3 Example Application

In this section, we describe an example application of our library—a verified SAT solver. The purpose of a SAT solver is to determine whether a given propositional formula is satisfiable. The Lean standard library defines a type of propositional formulae in conjunctive normal form (CNF), called `Std.Sat.CNF`. This `Std.Sat.CNF` type is indexed by another type α, which is the type of literals that appear in the formula. In our case, we set α to `Fin n`—the type of indices in a vector of size n. The standard library also provides a function `Std.Sat.CNF.eval`, which evaluates a given CNF formula with respect to a given assignment (a function mapping literals to Booleans, so `Fin n → Bool`) and outputs a Boolean. Lastly, the standard library provides a predicate `Std.Sat.CNF.Unsat`, which holds for a formula if it evaluates to `false` for all possible assignments.

In file `Sat.lean` in our library, we implement a decision procedure for this `Std.Sat.CNF.Unsat` predicate, which translates a given CNF φ to an ROBDD B_φ using the `BDD.and`, `BDD.or` and `BDD.not` functions, and finally applies `instDecidableSemanticEquiv` to check if $D(B) = \mathbf{0}$. By proving that our translation from CNF to BDD is semantically correct (which means that a formula φ evaluates to `true` with a given assignments if and only if $D(B_\varphi)$ yields `true` for the corresponding input), we are able to translate a proof that $D(B_\varphi) = \mathbf{0}$ back to a proof that φ is unsatisfiable, and likewise when we get $D(B_\varphi) \neq \mathbf{0}$ we are able to produce a proof that φ *is* satisfiable.

The following three functions implement the translation from CNF formulae to ROBDDs:

```
def BDD_of_literal (l : Std.Sat.Literal (Fin n)) : BDD :=
  if l.2 then (BDD.var l.1) else (BDD.var l.1).not
def BDD_of_clause (c : Std.Sat.CNF.Clause (Fin n)) : BDD :=
  (c.map BDD_of_literal).foldr BDD.or (BDD.const false)
def BDD_of_CNF (C : Std.Sat.CNF (Fin n)) : BDD :=
  (C.map BDD_of_clause).foldr BDD.and (BDD.const true)
```

Next, we establish that the input size of the resulting BDD is always bounded by the number of literals n:

```
lemma BDD_of_literal_nvars : (BDD_of_literal (n := n) C).nvars ≤ n := by
  simp only [BDD_of_literal]; split <;> (simp; omega)

lemma BDD_of_clause_nvars : (BDD_of_clause (n := n) C).nvars ≤ n := by
  induction C <;> simp_all [BDD_of_clause]

lemma BDD_of_CNF_nvars : (BDD_of_CNF (n := n) C).nvars ≤ n := by
  induction C <;> simp_all [BDD_of_CNF]
```

Lastly we prove that our translation from CNF to BDD is semantically correct, which allows us to define a `Decidable` instance for the `Std.Sat.CNF.Unsat` predicate, which goes through the translation to BDD and our `instDecidableSemanticEquiv` instance which decides if a given BDD is semantically equivalent to another BDD, in this case to `BDD.const false`:

```
lemma BDD_of_CNF_correct {f : Fin n → Bool} (C : Std.Sat.CNF (Fin n)) :
    Std.Sat.CNF.eval f C =
    (BDD_of_CNF C).denotation (n := n) (by simp) (Vector.ofFn f) := by
  induction C with
  | nil => simp [BDD_of_CNF]
  | cons head tail ih =>
    simp only [Std.Sat.CNF.eval_cons, BDD_of_CNF, ...]
    simp only [Std.Sat.CNF.eval_cons, BDD_of_CNF, ...] at ih
    rw [ih]
    congr 1
    induction head with
    | nil => simp [BDD_of_clause]
    | cons head tail ih =>
      simp only [Std.Sat.CNF.Clause.eval_cons, Fin.eta]; rw [ih]
      simp only [BDD_of_clause, Fin.eta, ...]
      congr 1
      simp only [BDD_of_literal]; split <;> simp_all

instance instDecidableUnsat (C : Std.Sat.CNF (Fin n)) : Decidable (Std.Sat.CNF.Unsat C) :=
  decidable_of_iff ((BDD_of_CNF C).SemanticEquiv (BDD.const false)) ⟨l_to_r, r_to_l⟩ where
    l_to_r h := by ...
    r_to_l h := by ...
```

To evaluate the performance of our SAT solver, we applied it to random 3-CNF formulae with differing numbers of variables and clauses, and measured the time that our SAT solver needed to check their satisfiability. All input CNF instances were generated using the CNFgen tool [Lau+17]. All performance measurements where conducted on a MacBook Pro laptop with a 2.3 GHz 8-Core Intel Core i9 CPU and 64GB of 2667 MHz DDR4 memory, using the `SatSolver.lean` program which can be found in the library's repository at https://github.com/eshelyaron/lean4-bdd.

For each input size n between 10 and 50, we checked 5 random 3-CNF formulae, each with 128 clauses, and measured the time each run took. We also measured the maximum BDD size created during the run, by slightly modifying `BDD_of_CNF` as presented above to keep track of the maximum BDD size while processing the CNF clauses. We found that both the run time and the maximum BDD size grow exponentially with the number of variables, as one would expect. Figure 2.2 shows these results, on the left we see the average run time on a logarithmic scale, plotted as a function of the number of variables; on the right we see the run time plotted as a function of the maximum BDD size with both axes scaled logarithmically. The right plot in particular shows that the run time of our SAT solver is roughly proportional to the maximum BDD size.

At around 50 variables, we reach maximum BDD size of the order of 10^8. Our library is able to handle such large BDDs, but constructing them takes a significant amount of time (several hours). Hence in modern SAT solver standards, our SAT solver performs rather poorly, seeing as state-of-the-art SAT solvers can often handle hundreds of thousands of variables and more [HJS10].

By applying a sampling profiler while running our SAT solver, we observe that most of the runtime is spent in the functions `Apply.apply_helper` and `Reduce.oreduce` which we have described in section 2.2.5. This finding accords with our expectations, since we expect most of the run time to be spent in the translation of CNF to BDD—where these two functions do most of the work—rather than in the check for satisfiability of the resulting BDD, which should be instantaneous.

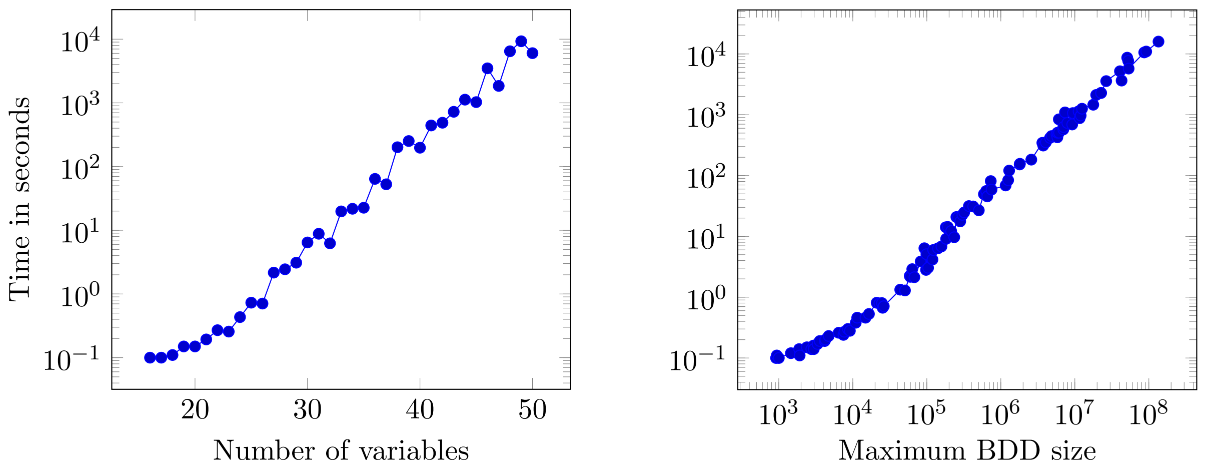

Figure 2.2: SAT solver run time by number of variables and maximum BDD size

In addition to varying the number of variables, we also measured the performance of our SAT solver with a varying number of 3-CNF clauses while keeping the number of variables fixed to 32. The run time of our SAT solver appears to be mostly independent of the number of clauses, as seen in fig. 2.3.

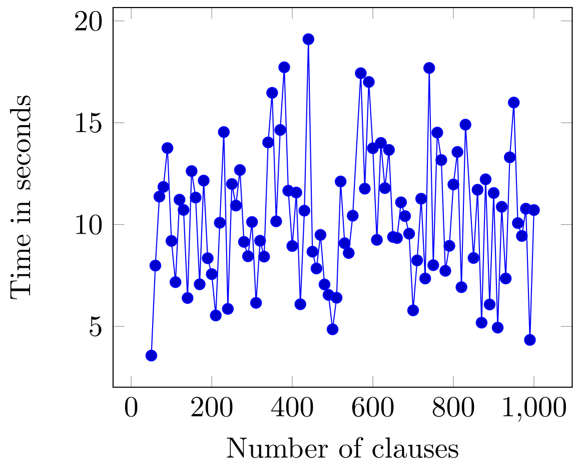

Figure 2.3: SAT solver run time by number of 3-CNF clauses

Chapter 3

Future Work and Summary

3.1 Directions for Further Development

In this section we discuss various ways in which our BDD library can be developed further.

Perhaps the most pressing task for our BDD library is to complete the correctness proof of our implementation of the *Reduce* algorithm, which we have discussed in section 2.2.5. This is the only part of the library that we have not fully formalized yet. Completing this last step would allow our library to be used in broader formalization projects and in applications that require strict correctness guarantees, as well as automated proof tactics in Lean based on the BDD-based decision procedures that our library provides.

We expect that completing the correctness proof for our current implementation of the *Reduce* algorithm will not be too difficult, based on our experience proving the correctness of the rest of the algorithms in the library. However, as we discuss below, there are also ways to enhance the current implementation, and such enhancements may affect any such correctness proof. Therefore an alternative approach would be to first optimize our implementation, e.g. by switching to a shared representation, and then to prove the correctness of the new implementation.

We would also like to extend our library with facilities for reasoning about the *sizes* of BDDs. Currently, we provide a function `BDD.size` which returns the size of a given BDD, but we do not provides any formal lemmas that facilitate reasoning about the result. The BDD sizes are of great interest since they significantly affect the performance of BDD based applications (as we saw, for example, in section 2.3), and indeed the problem of minimizing BDDs has been studied extensively in the BDD literature (e.g. [DG99; Fel+93; Sch+02; EGD03; BW96]).

In particular, we would like to formalize the notion of the *cofactors* of a Boolean function, and prove that the size of a BDD is lower-bounded by the number of its denotation's cofactors. This should be a rather straightforward task, and it would allow us to formalize the proof from [Hun97] which shows that the characteristic function for the set of permutations only admits BDDs of exponential size.

Next, we would like to implement and formally prove the correctness of BDD solution *counting* algorithm (see e.g. [Bry86; Knu09]). Currently, our library provides a method for obtaining one solution (an input for which a BDD's denotation returns `true`) via the `BDD.choice` and `BDD.find` functions. Solution counting yields the total number of such solutions, and can be achieved in $O(\text{size}(B))$ time for a BDD B, via a simple recursive algorithm. In [Knu09], Knuth shows several interesting combinatorial problems that can be solved via BDD-based solution counting, such as finding the number of independent sets in a given graph.

As discussed in section 2.1.3, our library provides the `BDD.apply` and `BDD.restrict` function which can be combined to implement all sorts operations on Boolean functions, such as existential and universal quantification. However, for some common operations, we can provide more efficient direct implementations, instead of building on top of `apply` and `restrict`. In particular, an efficient implementation of the *relational product* operation—for example, as described in [Yan+98]—can be very beneficial for model checking applications [Bry18; CG18].

Similarly, our library could be extended with a ternary *if-then-else* operator along the lines of [BRB91]. This is a common operation in BDD implementations in practice, provided by both CUDD and CacBDD, as two prominent examples [Som98; LSX13].

Lastly, we survey several implementation techniques that our library can adopt to improve performance.

First, our library can benefit from the use of *complemented edges* (CE) [Kar88; BRB91]. With CE, each BDD edge is enriched with a Boolean "flag" that says whether or not the edge is complemented. A complemented edge *negates* the denotation of the BDD it points to, so we can represent both a Boolean function f and its negation $\bar{f}$ using the same BDD. In particular, we only need one terminal node, since we can replace any edge pointing to the false terminal with a *complemented* edge pointing to true terminal. BDDs with CE are more compact, as they reuse the same space for Boolean functions that arise along with their negation, and they also let us negate a BDD in constant time (by (un)complementing one edge) whereas in our current implementation, without CE, applying negation to a BDD B takes $O(\text{size}(B))$ time.

We consider CE to be a natural next step in the development of our library, that should require no changes in the library's interface, and only some adjustments to the existing core definitions and proofs.

A more involved, yet very promising, potential enhancement for our library would be to represent all BDDs using one heap with multiple roots, which allows sharing common structures across different BDDs. Using such a *shared* representation [MIY91], all similar BDDs are represented by the same exact pointer, which reduces the task of checking for BDD similarity to mere a pointer equality test, which can be performed in constant time. (In our current implementation using a *split* representation, such a similarity check takes linear time, see section 2.2.4.)

Another prominent BDD optimization which our library currently lacks is *dynamic variable reordering*, which consists of switching the order of adjacent variables during the execution of a BDD-based program. Different variable orders induce BDDs of different sizes for a fixed Boolean function. Thus dynamic reordering techniques, such as the *sifting* algorithm ([Rud93]), can decrease the size of a constructed BDD, allowing our implementation to handle Boolean functions that it would otherwise fail to represent due to memory constraints.

In section 2.2.5, we mentioned a potential optimization for our implementation of the *Apply* algorithm that dates back to Bryant's description of the algorithm in [Bry86]: we can eliminate more recursive calls in `Apply.apply_helper` by leveraging the fact that the output of some binary Boolean operators (which the *Apply* algorithm applies to to two Boolean functions represented by BDDs) can be determined even if one or two of its arguments are not fully determined, e.g. $\mathsf{false} \cdot b = \mathsf{false}$ regardless of the value of b. Using complemented edges and a shared representation provides even more such early termination opportunities, as Bryant details in [Bry18].

Finally, the *Reduce* algorithm (section 2.2.5) involves a list sorting operation for which we currently rely on Lean's *merge sort* implementation in `List.mergeSort`. This has special runtime support in Lean that makes it especially efficient, but it nevertheless has $O(n \log n)$ time complexity. In [Knu09], Knuth shows a variant of the *Reduce* algorithm that employs a sorting method based on *bucket sort*, which has linear average time complexity.

When examining the performance our example SAT solver (section 2.3), we saw that `List.mergeSort` accounts for the second highest amount of total run time, right after `Apply.apply_helper`. Hence replacing `List.mergeSort` with a faster alternative may yield a significant performance improvement.

3.2 Conclusion

In this thesis, we developed and presented an implementation of a Binary Decision Diagrams library in Lean 4.

We began by presenting the theory of BDDs in chapter 1, followed by a detailed description of our library in chapter 2.

Notably, in section 2.1 we presented our library's interface, which provides methods for constructing BDDs and applying them to various tasks related to Boolean functions, such as efficiently deciding whether two BDDs represent the same Boolean function. The key characteristic of our library's interface is that it fully hides the details of our BDD implementation, allowing users to reason (prove statements) purely in terms of Boolean functions, while benefiting from the efficient algorithms that the underlying BDDs admit.

With the exception of one correctness proof which remains to be completed, our implementation is *fully verified*, in the sense that all of the library's functions are accompanied by correctness proofs in the formal system of Lean. These correctness proofs build on top of results from the theory of BDDs that we formalize and prove in our library, such as our proof that ROBDDs constitute a canonical form for Boolean functions (see section 2.2.4).

In section 2.2, we examined our implementation and formalization of BDDs in detail, and investigated some of the aforementioned formal correctness proofs, most notably the correctness proof for the BDD *Apply* algorithm which we discussed in section 2.2.5.

We evaluated an example application of our library, a BDD-based SAT solver, in section 2.3. This SAT solver has the unique advantage that it uses the correctness lemmas of our library to provide a Lean proof of (un)satisfiability as part of its output. However, its performance pales in comparison with state-of-the-art SAT solvers such as CaDiCal [Bie+24].

In section 3.1, we surveyed several possible enhancements that may improve our library's performance.

Bibliography

[And97] Henrik Reif Andersen. “An introduction to binary decision diagrams”. In: *Lecture notes, available online, IT University of Copenhagen* 5 (1997). http://cc.ee.ntu.edu.tw/~ric/teaching/DataStructureProgramming/S09/Project/Intro_BDD_Henrik97.pdf (cited on page 15).

[Bie+24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. “CaDiCaL 2.0”. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I.* Edited by Arie Gurfinkel and Vijay Ganesh. Volume 14681. Lecture Notes in Computer Science. Springer, 2024, pages 133–152. https://doi.org/10.1007/978-3-031-65627-9_7 (cited on page 69).

[BRB91] Karl S. Brace, Richard L Rudell, and Randal E Bryant. “Efficient implementation of a BDD package”. In: *Proceedings of the 27th ACM/IEEE design automation conference.* 1991, pages 40–45. https://doi.org/10.1145/123186.123222 (cited on pages 2, 68).

[Bry18] Randal E. Bryant. “Binary Decision Diagrams”. en. In: *Handbook of Model Checking.* Edited by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pages 191–217. ISBN: 978-3-319-10574-1. https://doi.org/10.1007/978-3-319-10575-8_7 (cited on pages 2, 15, 22, 27, 46, 68).

[Bry86] Randal E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *Computers, IEEE Transactions on* 100.8 (1986), pages 677–691. https://doi.org/10.1109/TC.1986.1676819 (cited on pages 2, 6, 9, 15, 20, 46, 54, 57, 58, 63, 67, 68).

[Bry95] Randal E. Bryant. “Binary decision diagrams and beyond: Enabling technologies for formal verification”. In: *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD).* IEEE. 1995, pages 236–243. https://doi.org/10.1109/ICCAD.1995.480018 (cited on page 2).

[BS21] Gregor Behnke and David Speck. “Symbolic search for optimal total-order HTN planning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Volume 35. 13. 2021, pages 11744–11754. https://doi.org/10.1609/aaai.v35i13.17396 (cited on page 2).

[Bur+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. “Symbolic model checking for sequential circuit verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (1994), pages 401–424. https://doi.org/10.1109/43.275352 (cited on pages 2, 27).

[BW96] Beate Bollig and Ingo Wegener. "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on computers* 45.9 (1996), pages 993–1002. https://doi.org/10.1109/12.537122 (cited on page 67).

[CG18] Sagar Chaki and Arie Gurfinkel. "BDD-Based Symbolic Model Checking". In: *Handbook of Model Checking*. Edited by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pages 219–245. ISBN: 978-3-319-10575-8. https://doi.org/10.1007/978-3-319-10575-8_8 (cited on pages 6, 27, 28, 68).

[DG99] Rolf Drechsler and Wolfgang Günther. "Using lower bounds during dynamic BDD minimization". In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 1999, pages 29–32. https://doi.org/10.1145/309847.309858 (cited on page 67).

[DS01] Rolf Drechsler and Detlef Sieling. "Binary decision diagrams in theory and practice". In: *International Journal on Software Tools for Technology Transfer* 3 (2001), pages 112–136. https://doi.org/10.1007/s100090100056 (cited on page 15).

[EGD03] Rüdiger Ebendt, Wolfgang Gunther, and Rolf Drechsler. "An improved branch and bound algorithm for exact BDD minimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.12 (2003), pages 1657–1663. https://doi.org/10.1109/TCAD.2003.819427 (cited on page 67).

[Fel+93] Eric Felt, Gary York, Robert Brayton, and Alberto Sangiovanni-Vincentelli. "Dynamic variable reordering for bdd minimization". In: *Proceedings of EURO-DAC 93 and EURO-VHDL 93-European Design Automation Conference*. IEEE. 1993, pages 130–135. https://doi.org/10.1109/EURDAC.1993.410627 (cited on page 67).

[Gat18] Malvin Gattinger. *New directions in model checking dynamic epistemic logic*. University of Amsterdam, 2018. https://malv.in/phdthesis (cited on pages 2, 27, 28).

[HJS10] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver". In: *Journal on Satisfiability, Boolean Modelling and Computation* 6.4 (2010), pages 245–262. https://doi.org/10.3233/SAT190070 (cited on page 66).

[Hun97] Ngai Ngai William Hung. "Exploiting symmetry for formal verification". Master's thesis. University of Texas at Austin, 1997. https://web.cecs.pdx.edu/~whung/papers/thesis.pdf (cited on pages 15, 67).

[Kar88] Kevin Karplus. *Representing Boolean functions with if-then-else DAGs*. University of California, Computer Research Laboratory, 1988. https://users.soe.ucsc.edu/~karplus/papers/ucsc-88-28.pdf (cited on page 68).

[Knu09] Donald E. Knuth. *The art of computer programming, volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams*. Addison-Wesley Professional, 2009. https://dl.acm.org/doi/abs/10.5555/1593023 (cited on pages 2, 5, 15, 29, 46, 63, 67, 69).

[Lau+17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. "CNFgen: A generator of crafted benchmarks". In: *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*. Springer. 2017, pages 464–473. https://doi.org/10.1007/978-3-319-66263-3_30 (cited on page 65).

[LSX13] Guanfeng Lv, Kaile Su, and Yanyan Xu. “CacBDD: A BDD package with dynamic cache management”. In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pages 229–234. https://doi.org/10.1007/978-3-642-39799-8_15 (cited on pages 2, 42, 68).

[MIY91] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. “Shared binary decision diagram with attributed edges for efficient Boolean function manipulation”. In: *Proceedings of the 27th ACM/IEEE design automation conference*. 1991, pages 52–57. https://doi.org/10.1145/123186.123225 (cited on pages 2, 68).

[MU21] Leonardo de Moura and Sebastian Ullrich. “The Lean 4 theorem prover and programming language”. In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pages 625–635. https://doi.org/10.1007/978-3-030-79876-5_37 (cited on pages 2, 16).

[Onl19] Stanford Online. *Stanford Lecture: Donald Knuth - ”Fun With Binary Decision Diagrams (BDDs)” (June 5, 2008)*. 2019. https://www.youtube.com/watch?v=SQE21efsf7Y&t=3878s (cited on page 2).

[Pau15] Christine Paulin-Mohring. “Introduction to the calculus of inductive constructions”. In: *All about Proofs, Proofs for All* 55 (2015). https://inria.hal.science/hal-01094195/ (cited on page 16).

[Rud93] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE. 1993, pages 42–47. https://doi.org/10.1109/ICCAD.1993.580029 (cited on pages 2, 68).

[Sch+02] Christoph Scholl, D Moller, Paul Molitor, and Rolf Drechsler. “BDD minimization using symmetries”. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 18.2 (2002), pages 81–100. https://doi.org/10.1109/43.743706 (cited on page 67).

[Sha38] Claude E. Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pages 713–723. https://doi.org/10.1109/EE.1938.6431064 (cited on page 6).

[Som98] Fabio Somenzi. “CUDD: CU decision diagram package release 2.3. 0”. In: *University of Colorado at Boulder* 621 (1998) (cited on pages 2, 42, 68).

[Tri16] Markus Triska. “The boolean constraint solver of SWI-Prolog (system description)”. In: *International Symposium on Functional and Logic Programming*. Springer. 2016, pages 45–61. https://doi.org/10.1007/978-3-319-29604-3_4 (cited on pages 3, 42).

[Ver+00] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S Arun-Kumar. “Reflecting bdds in coq”. In: *Annual Asian Computing Science Conference*. Springer. 2000, pages 162–181. https://doi.org/10.1007/3-540-44464-5_13 (cited on page 3).

[Yan+98] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. "A performance study of BDD-based model checking". In: *Formal Methods in Computer-Aided Design: Second International Conference, FMCAD'98 Palo Alto, CA, USA, November 4–6, 1998 Proceedings 2*. Springer. 1998, pages 255–289. https://doi.org/10.1007/3-540-49519-3_18 (cited on pages 27, 68).