Journal of Artificial Intelligence Research 69 (2020) 501–529 Submitted 06/2020; published 10/2020

Properties of Switch-List Representations of Boolean Functions

Ondřej Čepek ONDREJ.CEPEK@MFF.CUNI.CZ
Miloš Chromý MILOSCHROMY@GMAIL.COM
Dept. of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic

Abstract

In this paper, we focus on a less usual way to represent Boolean functions, namely on representations by switch-lists, which are closely related to interval representations. Given a truth table representation of a Boolean function f the switch-list representation of f is a list of Boolean vectors from the truth table which have a different function value than the preceding Boolean vector in the truth table. The main aim of this paper is to include this type of representation in the Knowledge Compilation Map by Darwiche and Marquis and to argue that switch-lists may in certain situations constitute a reasonable choice for a target language in knowledge compilation. First, we compare switch-list representations with a number of standard representations (such as CNF, DNF, and OBDD) with respect to their relative succinctness. As a by-product of this analysis, we also give a short proof of a long-standing open question proposed by Darwiche and Marquis, namely the incomparability of MODS (models) and PI (prime implicates) representations. Next, using the succinctness result between switch-lists and OBDDs, we develop a polynomial time compilation algorithm from switch-lists to OBDDs. Finally, we analyze which standard transformations and queries (those considered by Darwiche and Marquis) can be performed in polynomial time with respect to the size of the input if the input knowledge is represented by a switch-list. We show that this collection is very broad and the combination of polynomial time transformations and queries is quite unique. Some of the queries can be answered directly using the switch-list input, others require a compilation of the input to OBDD representations which are then used to answer the queries.

1. Introduction

A Boolean function on n variables is a mapping from $\{0,1\}^n$ to $\{0,1\}$. This concept naturally appears and is extensively used in several areas of mathematics and computer science. There are many different ways in which a Boolean function may be represented. Common representations include truth tables (TT) with 2^n rows (where a function value is explicitly given for every binary vector), list of models (MODS), i.e. a list of binary vectors on which the function evaluates to 1, various types of Boolean formulas (including CNF and DNF representations), various types of binary decision diagrams (BDDs, FBDDs, OBDDs), and negational normal forms (NNF, DNNF, d-DNNF).

The task of transforming one of the representations of a given function f into another representation of f (e.g. transforming a DNF representation into an OBDD or a DNNF into a CNF) is called knowledge compilation. For a comprehensive review paper on knowledge compilation see (Darwiche & Marquis, 2002), where a Knowledge Compilation Map

(KCM) is introduced. KCM systematically investigates different representation languages with respect to (1) their relative succinctness, (2) the complexity of common transformations, and (3) the complexity of common queries. The ***succinctness*** of representations roughly speaking describes how large the output representation in language B is with respect to the size of the input representation in language A when compiling from A to B. A precise definition of this notion will be given later in this text. Transformations include negation, conjunction, disjunction, conditioning, and forgetting. The complexity of such transformations may differ dramatically from trivial to NP-hard depending on the chosen representation language. The same is true for queries such as consistency check, validity check, clausal and sentential entailment, equivalence check, model counting, and model enumeration.

In (Le Berre, Marquis, Mengel, & Wallon, 2018) the authors included Pseudo-Boolean constraint (PBC) and Cardinality constraint (CARD) languages into KCM by showing succinctness relations among PBC, CARD, and languages already in the map, and by proving the complexity status of all queries and transformations introduced in (Darwiche & Marquis, 2002). In this paper, we aim at achieving exactly the same goal for switch-list representations.

We start by introducing the closely related interval representations. Let f be a Boolean function and let us fix some order of its n variables. The input binary vectors can be now thought of as binary numbers (with bits in the prescribed order) ranging from 0 to $2^n - 1$. An interval representation (IR) is an abbreviated MODS representation, where instead of writing out all the models, we write out only those models x (i.e. $f(x) = 1$) for which $f(x-1) = 0$ ($x-1$ is a non-model of f) and those models y for which $f(y+1) = 0$ ($y+1$ is a non-model of f). Function f is then represented by an ordered list of such pairs $[x, y]$ of integers, each pair specifying one interval of models. Note that $x = y$ for those pairs which represent an interval with a single model.

Interval representations of Boolean functions were introduced in (Schieber, Geist, & Zaks, 2005), where the input was considered to be a function represented by a single interval (two n-bit numbers x, y) and the output was a DNF representing the same Boolean function f on n variables, i.e. a function which is true exactly on binary vectors (numbers) from the interval $[x, y]$. This knowledge compilation task originated from the field of automatic generation of test patterns for hardware verification (Lewin, Fournier, Levinger, Roytman, & Shurek, 1995; Huang & Cheng, 1999). In fact, the paper (Schieber et al., 2005) achieves more than just finding some DNF representation of the input 1-interval function — it finds in polynomial time the shortest such DNF, where "shortest" means a DNF with the least number of terms. Thus (Schieber et al., 2005) combines a knowledge compilation problem (transforming an interval representation into a DNF representation) with a knowledge compression problem (finding the shortest DNF representation).

In (Čepek, Kronus, & Kučera, 2008) the reverse knowledge compilation problem was considered. Given a DNF, test if all models form a single interval under some permutation of variables, and in the affirmative case output the permutation and the two n-bit numbers defining the interval (note, that changing the order of variables may dramatically change the length of interval representations from $O(n)$ to $\Omega(2^n)$ — see Section 3 for examples). This problem can be easily shown to be co-NP hard in general (it contains tautology testing for DNFs as a subproblem), but was shown in (Čepek et al., 2008) to be polynomially solvable for tractable classes of DNFs (where tractable means that DNF falsifiability can be decided in polynomial time for the inputs from the given class). The algorithm presented in (Čepek et al., 2008) runs in $O(n\ell f(n,\ell))$ time, where n is the number of variables and ℓ the total number of literals in the input DNF, while $f(n,\ell)$ is the time complexity of falsifiability testing on a DNF on at most n variables with at most ℓ total literals. This algorithm serves as a recognition algorithm for 1-interval functions given by tractable DNFs. This result was later extended in (Kronus & Čepek, 2008) to monotone 2-interval functions, where an $O(\ell)$ recognition algorithm for the mentioned class was designed. Recently, these results were further extended to k-interval functions for arbitrary k (a function is k-interval if there exists a permutation of variables for which the interval representation consists of at most k intervals). Paper (Čepek & Hušek, 2017) presents a recognition algorithm that runs in polynomial time in the length of the input DNF for any constant k (the complexity is exponential in k).

In fact, (Čepek & Hušek, 2017) departs from interval representations and introduces closely related switch-list representations which we shall use in this paper. Given a fixed order of variables of function f, a ***switch*** of f is a vector (binary number) x such that $f(x-1) \neq f(x)$. A ***switch-list*** is an ordered list of all switches of a given function. A switch-list of f together with the function value $f(0,0,\ldots,0)$ forms a ***switch-list representation (SLR)*** of f. It is important to note that switch-lists are ordered by the natural order on binary numbers (as opposed to maintaining sets of switches) as this helps to lower the complexity of several query and transformation algorithms described later in this paper.

SLRs have an added advantage over the IRs, namely that a function and its negation have the same switch-lists and the two representations differ only by the opposite values of $f(0,0,\ldots,0)$. The ease of taking a negation for SLRs allows a trivial translation of any result relating SLR and DNF to a result relating SLR and CNF (and vice versa). This is due to the fact that given a DNF, its logical negation can be represented by a CNF of the same length (and vice versa), and the transformation is purely mechanical (replace disjunctions by conjunctions, conjunctions by disjunctions, and negate all literals). Taking a negation is not as straightforward for interval representations, where the interval representations of a function and its negation may significantly differ, and even the number of intervals may be different (although the difference is at most one). For this reason, we shall use SLRs throughout this paper. It is not a limiting assumption in any way: clearly, the list of intervals can be easily compiled in linear time from the list of switches and the function value $f(0,0,\ldots,0)$, and vice versa. In fact, we shall use this compilation between SLRs and interval IRs later in this paper, since some algorithms (e.g. the one for the forgetting transformation) are more naturally designed for IRs than for SLRs.

The languages of SLRs and IRs may be in some situations quite a good choice as a target compilation language. In the succinctness map these languages are placed strictly above TT and MODS, incomparable to prime implicates (PI) and prime implicants (IP), and strictly below CNF, DNF, and OBDD languages. However, compared to CNF, DNF, and OBDD (and even IP and PI) they have a wider set of supported queries and transformations.

SLRs support all the queries from (Darwiche & Marquis, 2002) in polynomial time (see Table 1), which is of course better than CNFs and DNFs but also better than IP and PI which do not support model counting. It is also better than general OBDDs which do not support sentential entailment if the two input OBDDs respect different orders of variables.

The only language considered in (Darwiche & Marquis, 2002) with the same set of supported queries is the language of OBDDs with a fixed order of variables. Hence, the advantage of SLR is that it does not require the same order of variables for all inputs to guarantee polynomial time bounds on all queries. Moreover, an added advantage of SLRs lies in the computational simplicity of answering most queries. Validity and consistency checks are trivial (constant time), clausal entailment, implicant check, and model counting take linear time (w.r.t. the input size), and model enumeration takes linear time w.r.t. the output size. The only queries that are time-consuming are sentential entailment and equivalence check. We do not have direct algorithms that manipulate SLRs for these queries at the present moment. Instead, we compile both input SLRs into OBDDs (both respecting the same order of variables) and run the query algorithms for these OBDDs. The polynomial time compilation algorithm from a SLR to an OBDD with different variable orders on the input and on the output is one of the main contributions of this paper. Finding direct algorithms for sentential entailment and equivalence check which would avoid compilation into OBDDs may be a good research topic.

L	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
NNF	○	○	○	○	○	○	○	○
DNNF	✓	○	✓	○	○	○	○	✓
d-NNF	○	○	○	○	○	○	○	○
BDD	○	○	○	○	○	○	○	○
FBDD	✓	✓	✓	?	○	○	✓	✓
OBDD	✓	✓	✓	✓	✓	○	✓	✓
OBDD$_<$	✓	✓	✓	✓	✓	✓	✓	✓
CNF	○	✓	○	✓	○	○	○	○
DNF	✓	○	✓	○	○	○	○	✓
IP	✓	✓	✓	✓	✓	✓	○	✓
MODS	✓	✓	✓	✓	✓	✓	✓	✓
CARD	○	✓	○	✓	○	○	○	○
PBC	○	✓	○	✓	○	○	○	○
SL	✓	✓	✓	✓	✓	✓	✓	✓
SL$_<$	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Languages introduced by (Darwiche & Marquis, 2002) and (Le Berre et al., 2018) and their polytime queries. ✓means "satisfies" and ○ means "does not satisfy unless P=NP". Languages **SL** and **SL**$_<$ are defined in Section 2

The biggest advantage of SLRs over the strictly more succinct representations such as CNFs, DNFs, and OBDDs rests in the collection of supported transformations (see Table 2). SLRs support negation (in constant time) unlike CNFs, DNFs, and MODS, for which the size of negation can grow exponentially. SLRs also support conditioning (but all common representations do, so this is not an advantage) and more importantly forgetting (the general case, not just singleton forgetting) which distinguishes it from OBDDs that do not support forgetting. SLRs also support unbounded conjunction and disjunction under the additional restriction that all conjuncts (disjuncts) are defined on the same set of variables and with the same order of variables, i.e. all switches are vectors of the same length with individual coordinates indexed by the same variables for all input SLRs (this is not shown in Table 2 where only the general forms of conjunction and disjunction are tabulated). It should be noted here that OBDDs and even OBDDs with prescribed variable order fail to support unbounded conjunction and disjunction even in this restricted case. Of course, DNFs do not support unbounded conjunction, and CNFs do not support unbounded disjunction. If we allow different conjuncts (disjuncts) to be defined on different sets of variables, the output SLR may grow exponentially if we prescribe a variable order on the output. For the non-prescribed order of variables (on the output), we have no results at all, the complexity of conjunction and disjunction is open in this case.

L	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
DNNF	✓	✓	✓	○	○	✓	✓	○
d-NNF	✓	○	✓	✓	✓	✓	✓	✓
BDD	✓	○	✓	✓	✓	✓	✓	✓
FBDD	✓	●	○	●	○	●	○	✓
OBDD	✓	●	✓	●	○	●	○	✓
OBDD$_<$	✓	●	✓	●	✓	●	✓	✓
CNF	✓	○	✓	✓	✓	●	✓	●
DNF	✓	✓	✓	●	✓	✓	✓	●
IP	✓	●	●	●	✓	●	●	●
MODS	✓	✓	✓	●	✓	●	●	●
CARD	✓	○	?	✓	✓	●	●	●
PBC	✓	●	●	✓	✓	●	●	●
SL	✓	✓	✓	?	?	?	?	✓
SL$_<$	✓	✓	✓	●	●	●	●	✓

Table 2: Classes introduced by (Darwiche & Marquis, 2002) and (Le Berre et al., 2018) and their polytime transformations. ✓means "satisfies", ● means "does not satisfy" and ○ means "does not satisfy unles P=NP". Languages **SL** and **SL**$_<$ are defined in Section 2

The collection of supported queries and transformations suggests that SLRs may be a good choice in cases when many queries (such as model counting) have to be answered under many different additional assumptions such as a partial substitution of binary values to subsets of variables (i.e. conditioning) or existential quantification of subsets of variables (i.e. forgetting). None of the above mentioned more succinct representation would support such a scenario in polynomial time. An obvious problem for this approach is a lack of compilation algorithms with SLR as the target representation. The only interesting example is the recognition algorithm from (Čepek & Hušek, 2017), which may be in this context viewed as a compilation algorithm from tractable classes of DNFs into SLRs (and also from tractable CNFs by taking a negation of the input, compiling into a switch-list and then taking a negation on the output). Note that the algorithm has a parameter k on its input, and the compilation which runs in time exponential in k is successful if and only if there exists a target SLR with at most k switches. Another representation that is also easy to compile into a SLR is a binary decision tree with a fixed order of variables on all branches. By traversing the leaves of such a tree from left to right one can easily construct a SLR of the given function. This procedure is in some sense just the reverse of the algorithm presented in Section 4.1.

This is a theory paper that establishes the properties of SLRs and places the languages based on SLRs into KCM. We believe that the proven properties of SLRs will motivate an interest to find other classes of CNF, DNF, OBDD, or other representations, which can be efficiently compiled into SLRs.

As a final remark let us note that the combination of results from (Čepek et al., 2008) and (Schieber et al., 2005) gives a polynomial time minimization (optimal compression) algorithm for the class of 1-interval functions given by tractable DNFs, or in other words, for the 1-interval subclass of DNFs inside any tractable class of DNFs. DNF minimization (optimal compression) is a notoriously hard problem. It was shown to be Σ_2^p-complete (Umans, 2001) when there is no restriction on the input DNF (see also the review paper (Umans, Villa, & Sangiovanni-Vincentelli, 2006) for related results). It is also long known that this problem is NP-hard already for some tractable classes of DNFs — maybe the best known example is the class of Horn DNFs (a DNF is Horn if every term in it contains at most one negative literal) for which the NP-hardness was proved in (Ausiello, D'Atri, & Sacca, 1986; Hammer & Kogan, 1993) and the same result for cubic Horn DNFs in (Boros, Čepek, & Kučera, 2013). There exists a hierarchy of subclasses of Horn DNFs for which there are polynomial time minimization algorithms, namely acyclic and quasi-acyclic Horn DNFs (Hammer & Kogan, 1995), and CQ Horn DNFs (Boros, Čepek, Kogan, & Kučera, 2009). Suppose we are given a Horn DNF. We can test in polynomial time using the algorithm from (Čepek et al., 2008) whether it represents a 1-interval function and then (in the affirmative case) use the algorithm from (Schieber et al., 2005) to construct a minimum DNF representing the same function as the input DNF. Thus we have a minimization algorithm for 1-interval Horn DNFs. It is an interesting research question in what relation (with respect to inclusion) is this class with respect to the already known hierarchy of polynomial time compressible subclasses of Horn DNFs (acyclic Horn, quasi-acyclic Horn, and CQ Horn DNFs).

The paper is organized as follows. In Section 2 we shall introduce the necessary terminology and notation, and define the propositional languages studied in this paper. Section 3 derives the succinctness relations for the defined languages. Section 4 provides a compilation algorithm from SLRs to OBDD representations respecting different prescribed orders of variables on the input and on the output. In Section 5 we will prove a lower bound on the size of the compiled OBDD. Next, in Section 6 we investigate the complexity of common transformations for SLRs, and finally in Section 7 we do the same for common queries. We finish the paper with a few concluding remarks.

This is a full version of a conference paper (Čepek & Chromý, 2020) which also includes the compilation algorithm first presented in (Čepek & Chromý, 2020). This full version contains all proofs omitted or shortened in (Čepek & Chromý, 2020) as well as an additional material skipped there because of the page limit. Compared to (Čepek & Chromý, 2020) it also studies a second representation language based on SLRs.

2. Definitions and Notation

A ***Boolean function***, or a ***function*** in short, in n propositional variables is a mapping $f : \{0,1\}^n \to \{0,1\}$, where $x \in \{0,1\}^n$ is called a ***Boolean vector*** (a ***vector*** in short). A function f in n variables can be represented by a truth table, which is a list of all 2^n vectors together with their function values. Rather than listing all vectors, one can list only models of f (all vectors x for which $f(x) = 1$). The language of all such representations of all Boolean functions is called **MODS**[1], and each list of models for a particular function is called a ***sentence*** of the **MODS** language. Similarly we can consider sentences of non-models (all vectors x for which $f(x) = 0$) which define the language ¬**MODS** used in this paper for symmetry purposes.

Propositional variables $x_1, x_2, \dots$ and their negations $\neg x_1, \neg x_2, \dots$ are called ***literals*** (***positive*** and ***negative literals*** respectively). An elementary conjunction of literals

$$T = \bigwedge_{i \in I} x_i \wedge \bigwedge_{j \in J} \neg x_j \tag{1}$$

is called a ***term***, if every propositional variable appears in it at most once, i.e. if $I \cap J = \emptyset$. A ***disjunctive normal form*** (or DNF) is a disjunction of terms. Similarly, an elementary disjunction of literals

$$C = \bigvee_{i \in I} x_i \vee \bigvee_{j \in J} \neg x_j \tag{2}$$

is called a ***clause***, if every propositional variable appears in it at most once, i.e. if $I \cap J = \emptyset$. A ***conjunctive normal form*** (or CNF) is a conjunction of clauses.

It is a well-known fact, that every Boolean function can be represented by DNFs and CNFs (typically by many different ones). By **DNF** we shall denote the propositional language of all DNFs (of all functions), and similarly **CNF** shall denote the language of all CNFs. Each individual DNF (or CNF respectively) is then called a ***sentence*** of the **DNF** (or **CNF** respectively) language.

Given Boolean functions f and g on the same set of variables $\{x_1, \dots, x_n\}$, we denote by $f \leq g$ the fact that $f(x_1, \dots, x_n) \leq g(x_1, \dots, x_n)$ for every vector $(x_1, \dots, x_n)$, i.e. that g is satisfied for any assignment of values to the variables for which f is satisfied. Hence, for example, if a term t consists of a subset of literals which constitute term t' then $t' \leq t$. Similarly, if a clause c consists of a subset of literals which constitute clause c' then $c \leq c'$.

We call a term t an ***implicant*** of a function f, if $t \leq f$. Given DNF F representing function f it is obvious that every term $t \in F$ is an implicant of f, but there are typically many other implicants of f which are not explicitly present in F. We call t a ***prime implicant*** of f, if t is an implicant of f and there is no implicant $t' \neq t$ of f, for which $t \leq t' \leq f$. We call DNF F representing f ***canonical***, if it consists exactly of all prime implicants of f. The language **IP** is the subset of **DNF** where each sentence is a canonical DNF. Note, that unlike in the **DNF** language, each Boolean function is represented by exactly one sentence in the **IP** language.

1. It should be noted that the **MODS** language is defined as a subset of the **DNF** language in (Darwiche & Marquis, 2002), namely as the subset of all deterministic and smooth sentences. However, as long as no irrelevant variables are considered, both definitions are equivalent with an obvious bijection between models (binary vectors of length n) and terms (conjunctions of literals of length n).

Symmetrically, we call a clause c an ***implicate*** of a function f, if $f \leq c$. We call c a ***prime implicate*** of f, if c is an implicate of f and there is no implicate $c' \neq c$ of f, for which $f \leq c' \leq c$. We call CNF F representing f ***canonical***, if it consists exactly of all prime implicates of f. The language **PI** is the subset of **CNF** where each sentence is a canonical CNF.

A ***Binary Decision Diagram (BDD)*** is a rooted directed graph with two terminals labeled 0 and 1. Each non-terminal node is a decision node with exactly two outgoing edges. Each decision node corresponds to a propositional variable and the two outgoing edges correspond to the assignments of 0 and 1 to this variable. Each directed path from the root to a terminal thus corresponds to a (possibly partial) assignment of truth values to variables and the terminal specifies the function value for such an assignment. Let $<$ be a total order on the set PS of all propositional variables (we assume this set to be denumerable). An ***Ordered Binary Decision Diagram (OBDD)*** with respect to $<$ is a BDD such that on every path from the root to a terminal no two decision nodes correspond to the same variable and moreover every such path respects the prescribed order $<$. The second condition means that there does not exist a directed path p from the root to a terminal and two variables $x < y$, such that the decision node corresponding to y precedes the decision node corresponding to x on path p. All OBDDs with respect to $<$ form the language $\mathbf{OBDD}_<$ and the language **OBDD** is defined as the union of $\mathbf{OBDD}_<$ languages over all total orders on the set PS.

A ***Binary Decision Tree (BDT)*** is a rooted tree in that each internal node is a decision node and every leaf represents a terminal labeled either 0 or 1. By merging all leaves with label 0 to a single terminal and similarly for all leaves with label 1, we get a special type of a BDD where each decision node (except the root) has exactly one incoming edge.

Now we are ready to define the two principal languages of this paper.

Definition 2.1. *Let $<$ be a total order on the set PS of all propositional variables, let X be a subset of PS of size n, and let f be a Boolean function on variables from X. Consider vector $x \in \{0,1\}^n$ where the bits of x correspond to the variables of X in the prescribed order $<$. Each such vector x can be in natural way identified with a binary number from $[0, 2^n - 1]$, so for every $x > 0$ the vector $x-1$ is well defined. We call $x \in \{0,1\}^n$ a* **switch** *of f with respect to order $<$, if $f(x-1) \neq f(x)$. The list of all switches of f with respect to $<$ is called the* **switch-list** *of f with respect to $<$. The switch-list of f with respect to $<$ together with the function value $f(0)$ is called the* **switch-list representation (SLR)** *of f with respect to $<$. The set of switch-list representations with respect to $<$ (of all functions) forms the propositional language $\mathbf{SL}_<$. Finally, the language* **SL** *is the union of $\mathbf{SL}_<$ languages over all total orders on the set PS.*

Let us define the following two notions. Function f is called a ***k-switch function*** if there exists a SLR of f with respect to some order $<$ of its variables that has at most k switches. Two sentences (possibly from two different propositional languages) are called ***logically equivalent*** if they represent the same function. Now we are ready to define the most important concept of the next section.

Definition 2.2. *A propositional language $\mathbf{L}$ is* **at least as succinct** *as a propositional language $\mathbf{K}$, denoted $\mathbf{L} \leq \mathbf{K}$, if and only if there exists a polynomial p such that for every* *sentence $\alpha \in \mathbf{K}$ there exists a logically equivalent sentence $\beta \in \mathbf{L}$ such that $|\beta| \leq p(|\alpha|)$ (where the size of a sentence is the number of bits necessary to encode it). If $\mathbf{L} \leq \mathbf{K}$ holds and $\mathbf{K} \leq \mathbf{L}$ does not (denoted $\mathbf{K} \not\leq \mathbf{L}$), we write $\mathbf{L} < \mathbf{K}$.*

The diagram in Figure 1, summarizing the succinctness relations of many commonly used propositional languages, appeared in (Darwiche & Marquis, 2002). It is amended here by the results from (Le Berre et al., 2018) dealing with the **PBC** and **CARD** languages. The main aim of the next section is to add the languages $\mathbf{SL}_<$ and **SL** into the framed part of the diagram in Figure 1 by establishing the succinctness relations to the languages already presented there.

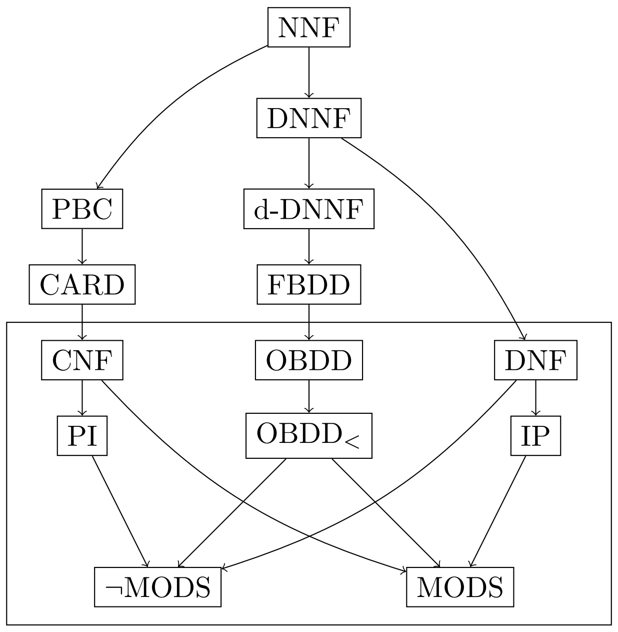


Figure 1: The diagram of succinctness relations from (Darwiche & Marquis, 2002) combined with the results from (Le Berre et al., 2018). For symmetry reasons the language ¬**MODS** was added into the diagram. Each directed arc $A \longrightarrow B$ means that A is strictly more succinct than B, i.e. $A < B$.

3. Succinctness of Switch-List Representations

In this section we prove the succinctness relations for $\mathbf{SL}_<$ and **SL** languages described in Figure 2. We will use one subsection for each relation, the numbering of the subsections corresponds to the arrow numbers in Figure 2. Let us start with the most obvious of all the relations.

3.1 Relation between SL and $\mathbf{SL}_<$ Languages

Proposition 3.1. $\mathbf{SL} < \mathbf{SL}_<$

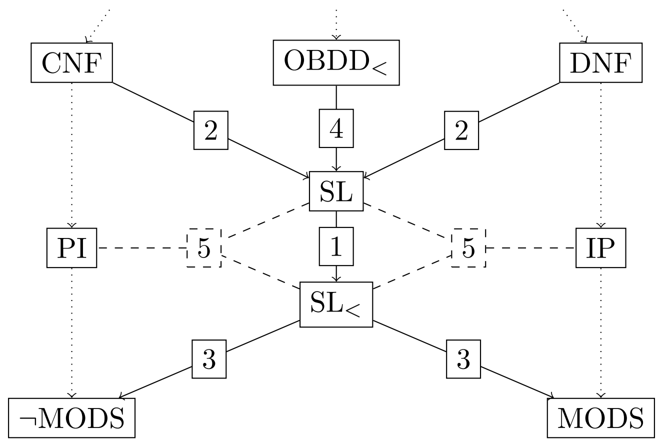


Figure 2: Solid arrows correspond to strict succinctness results, dashed lines to incomparability results, and dotted lines to known strict succinctness relations from Figure 1 which do not follow from transitivity using the solid arrows. The numbers on arrows and lines correspond to subsection numbers (of Section 3) in which the corresponding result is proved.

Proof. The inequality $\mathbf{SL} \leq \mathbf{SL}_<$ follows from the fact that the language $\mathbf{SL}_<$ is a subset of the language $\mathbf{SL}$. To show that $\mathbf{SL}_< \not\leq \mathbf{SL}$ let us consider function $f(x_1, \dots, x_n, y) = y$ and two orders of its variables $<_1$ and $<_2$ where $y <_1 x_1 <_1 \dots <_1 x_n$ and $x_1 <_2 \dots <_2 x_n <_2 y$. Clearly, the SLR of f with respect to $<_1$ has a single switch (all non-models precede all models) while the SLR of f with respect to $<_2$ has $2^{n+1} - 1$ switches (non-models and models alternate, so every vector except of the very first one is a switch). Thus the SLR of f in the language $\mathbf{SL}_{<_2}$ is exponentially larger than the SLR of f with respect to $<_1$ in the language $\mathbf{SL}$. This simple example is not fully satisfactory, because f does not depend on variables $x_1, \dots, x_n$ and so $f(x_1, \dots, x_n, y) = f(y)$ is in fact a function in just one variable (and of course the exponential blowup disappears). This can be easily fixed by switching the last non-model of f into a model, i.e. by considering $f(x_1, \dots, x_n, y) = y \vee (\bigwedge_{i=1}^n x_i)$. This function now depends on all $n+1$ variables, still has a single switch with respect to $<_1$ (all non-models still precede all models), and still has exponential many switches with respect to $<_2$ (only the last two switches with respect to $<_2$ disappeared). $\square$

3.2 Relations among SL, CNF, and DNF Languages

Proposition 3.2. $\mathbf{CNF} < \mathbf{SL}$ *and* $\mathbf{DNF} < \mathbf{SL}$

Proof. Let us start by proving $\mathbf{DNF} \leq \mathbf{SL}$. It was shown in (Schieber et al., 2005) that any 1-interval function on n variables can be represented by a DNF with at most $2n - 4$ terms. Obviously, the output DNF has a size which is at most quadratic in n and hence also at most quadratic in the size of the input SLR (two vectors of length n). Now assume we have SLR of function f with k switches on the input, which means that f has $\lfloor k/2 \rfloor$ or $\lfloor k/2 \rfloor + 1$ intervals of models (depending on the parity of k and the function value f(0)). Let us construct a DNF representation for each interval using the algorithm from (Schieber et al., 2005). Obviously, the disjunction of these DNFs represents f and the size of this aggregated DNF is $O(kn^2)$ while the size of the input is $O(kn)$.

Now let us show **CNF** $\leq$ **SL**. Starting with the SLR of a function f with k switches we can turn it in a constant time into the SLR of $\neg f$ by negating the value of $f(0)$ and keeping the switch-list unchanged. Both SLRs have size $O(kn)$. Using the construction from the previous paragraph, we get a DNF of $\neg f$ of size $O(kn^2)$. This DNF can be switched in $O(kn^2)$ time into a CNF of the same size which represents f (by mechanically applying de Morgan rules to propagate the negation on the outside of the DNF formula towards the literals, which changes the DNF into a CNF).

In order to prove **SL** $\not\leq$ **DNF** we shall assume by contradiction that **SL** $\leq$ **DNF**. Note, that by the transitivity of succinctness relations and the above proved relation **CNF** $\leq$ **SL** we could conclude **CNF** $\leq$ **DNF**, which is known to be false (see e.g. (Darwiche & Marquis, 2002) for a counterexample). The proof of **SL** $\not\leq$ **CNF** proceeds in a completely similar way, only the languages **CNF** and **DNF** exchange their roles. □

3.3 Relations among $\mathbf{SL}_<$, MODS, and ¬MODS Languages

Proposition 3.3. $\mathbf{SL}_< < \mathbf{MODS}$ *and* $\mathbf{SL}_< < \neg\mathbf{MODS}$

Proof. Consider the function $f(x_1, \ldots, x_n, y) = y \vee (\bigwedge_{i=1}^{n} x_i)$ from Proposition 3.1. It depends on all $n+1$ variables, has a single switch with respect to order $y < x_1 < \ldots < x_n$ of the variables, and has $2^n + 1$ models and $2^n - 1$ non-models. This proves $\mathbf{MODS} \not\leq \mathbf{SL}_<$ and $\neg\mathbf{MODS} \not\leq \mathbf{SL}_<$.

The relations $\mathbf{SL}_< \leq \mathbf{MODS}$ and $\mathbf{SL}_< \leq \neg\mathbf{MODS}$ are more or less obvious. It is easy to see that there can be at most twice as many switches as models and symmetrically also at most twice as many switches as non-models. Indeed, every switch x can be identified with a pair of consecutive vectors x and $x-1$ with opposite function values, and every model (and non-model) is identified in this way with at most two switches. □

3.4 Relation between SL and $\mathbf{OBDD}_<$ Languages

Let us start this subsection with several definitions and technical lemmas.

Definition 3.4. *Let $f(x_1, \ldots, x_n)$ be a Boolean function. A Boolean vector $(y_1, \ldots, y_l) \in \{0,1\}^l$ where $l < n$ is called a* **relevant vector for** f **of length** l *if there exist two vectors $(x_{l+1}, \ldots, x_n) \in \{0,1\}^{n-l}$ and $(x'_{l+1}, \ldots, x'_n) \in \{0,1\}^{n-l}$ such that*

$$f(y_1, \ldots, y_l, x_{l+1}, \ldots, x_n) \neq f(y_1, \ldots, y_l, x'_{l+1}, \ldots, x'_n)$$

Relevant vectors are prefix vectors (for the given order of variables) which are not sufficient for determining the value of f. The motivation behind the above definition is the easily verifiable fact that a k-switch function (i.e. function which has k switches with respect to the prescribed order of variables) has at most k relevant vectors of any length.

Lemma 3.5. *Let $f(x_1, \ldots, x_n)$ be a k-switch function and $1 \leq l \leq n$ an arbitrary number. Then there are at most k relevant vectors for f of length l.*

Proof. Let $y^i = (y^i_1, \ldots, y^i_n)$, $1 \leq i \leq k$ be the switch vectors for f and let us assume these vectors are lexicographically ordered. That is $\forall i \in \{1, \ldots, k-1\} : y^i < y^{i+1}$ if we identify switch vectors with binary numbers. Consider the vectors $p^i = (y_1^i, \dots, y_l^i)$, $1 \le i \le k$, that is the prefixes of the switch vectors of length l. There are at most k distinct vectors in this set as some pairs of prefixes may coincide. We claim that no other vector (different from $p^1, \dots, p^k$) is a relevant vector for f of length l. Let $z = (z_1, \dots, z_l)$ be any such vector. Since z differs from all p^i's, there is no switch vector among vectors $(z_1, \dots, z_l, x_{l+1}, \dots, x_n)$ for $(x_{l+1}, \dots, x_n) \in \{0,1\}^{n-l}$ and thus

$$f(z_1, \dots, z_l, x_{l+1}, \dots, x_n) = f(z_1, \dots, z_l, x'_{l+1}, \dots, x'_n)$$

for any two vectors $(x_{l+1}, \dots, x_n) \in \{0,1\}^{n-l}$ and $(x'_{l+1}, \dots, x'_n) \in \{0,1\}^{n-l}$, which proves that z is not relevant for f. □

Remark 3.6. *Note that in the above proof vectors $p^1, \dots, p^k$ are the only candidates for relevant vectors of length l, however not all of them have to be relevant for f, since the "decision" determining the value of f may be taken at an earlier index than l. A trivial example is a 1-switch function where $f(0, x_2, \dots, x_n) = 0$ and $f(1, x_2, \dots, x_n) = 1$ for all vectors $(x_2, \dots, x_n)$. Here no (non-empty) prefix of the single switch vector $(1, 0, 0, \dots, 0)$ is a relevant vector for f.*

Definition 3.7. *Let $f(x_1, \dots, x_n)$ be a Boolean function. A relevant vector $(y_1, \dots, y_l)$ for f of length l is called* **maximal relevant for** *f if neither $(y_1, \dots, y_l, 0)$ nor $(y_1, \dots, y_l, 1)$ are relevant vectors for f of length $l + 1$.*

Lemma 3.8. *Let f be a k-switch function. Then there are at most k maximal relevant vectors for f.*

Proof. As we have seen in the proof of Lemma 3.5, only prefixes of the switch vectors are candidates to relevant vectors. Moreover for each switch vector at most one length of prefix can be a maximal relevant length, which proves the claim. (Note that for some switch vectors no prefix is relevant so there is also no maximal relevant prefix - see Remark 3.6.) □

Lemma 3.9. *Let $f(x_1, \dots, x_n)$ be a Boolean function and $I \subseteq \{x_1, \dots, x_n\}$ be a subset of variables of size $|I| = i$. Let x_m be the variable with the smallest index in I and let us assume that f has at most k maximal relevant vectors for f of length at least m. Then there are at most $(ik + 1)$ different Boolean functions that originate from f by fixing the values of variables in I.*

Proof. If $k = 0$, that is there is no relevant vector for f of length m or larger, then f does not depend on variables from I. That means that all substitutions for the variables of I lead to the same function of $(n - i)$ variables and the claim holds $((ik + 1) = (i \cdot 0 + 1) = 1)$.

If $k \ge 1$ we shall proceed by induction on $|I| = i$.

Base case $(i = 1)$***:***

By fixing the only variable in $I \subseteq \{x_1, \dots, x_n\}$ to 0 or 1 we get at most two different functions of $n - 1$ variables, namely $f(x_1, \dots, x_{m-1}, 0, x_{m+1}, \dots, x_n)$ and $f(x_1, \dots, x_{m-1}, 1, x_{m+1}, \dots, x_n)$. Since $i = 1$ and $k \ge 1$ we get $ik + 1 \ge 2$ and the base case is verified.

Induction step $(i - 1 \Longrightarrow i)$***:***

Let us assume that $i > 1$ and the statement of the lemma is true for $1, 2, \dots, i - 1$. Let

$$V = \{p^i | 1 \le i \le l\}$$

be the set of all maximal relevant vectors for f of length at least m (by assumption $l \leq k$). Consider the partition of V depending on the value of x_m into

$$V_0 = \{p^i | p^i_m = 0\}$$
$$V_1 = \{p^i | p^i_m = 1\}$$

and denote $|V_0| = l_0$ and $|V_1| = l_1$. Clearly $l_0 + l_1 = l$.

Denote $I' = I \setminus \{x_m\}$ and let $x_{m'}$ be the variable with the smallest index in I' (of course $m' > m$). Consider functions of $(n-1)$ variables

$$f_0(x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n) = f_{|x_m=0}$$
$$f_1(x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n) = f_{|x_m=1}$$

There are at most l_0 maximal relevant vectors for f_0 of length at least m' (and similarly for f_1). Indeed, such maximal relevant vectors can originate from vectors in V_0 (or V_1 respectively) by deleting p^i_m, if those vectors are long enough (have length at least m'). Thus we may use the induction hypothesis for f_0, f_1 and I' of size $|I'| = i - 1$. We get that

1. there are at most $(i-1)l_0 + 1$ different Boolean functions that originate from f_0 by fixing the values of variables in I'
2. there are at most $(i-1)l_1 + 1$ different Boolean functions that originate from f_1 by fixing the values of variables in I'

This altogether implies that there are at most

$$\begin{aligned}&(i-1)l_0 + 1 + (i-1)l_1 + 1 = (i-1)(l_0 + l_1) + 2 = \\ &= (i-1)l + 2 \leq (i-1)k + 2 = ik - k + 2 \leq ik + 1\end{aligned}$$

different Boolean functions that originate from f by fixing the values of variables in I, since $k \geq 1$, which finishes the proof. □

Corollary 3.10. *Let $f(x_1, \ldots, x_n)$ be a k-switch Boolean function and $I \subseteq \{x_1, \ldots, x_n\}$ a subset of its variables of size $|I| = i$. Then there are at most $(ik + 1)$ different functions that originate from f by fixing variables in I.*

Proof. This claim is direct consequence of Lemma 3.8 and Lemma 3.9 □

Let us consider a k-switch function $f(x_1, \ldots, x_n)$, and let us consider an arbitrary reordering of the variables given by some linear order $<$. If we start branching on variables in the order given by $<$, then Corollary 3.10 states that after branching on the first i variables, we get at most $(ik + 1)$ different Boolean functions of the remaining variables (as opposed to at most 2^i for general functions). This is sufficient for a bound on the size of a minimal OBDD representation of f due to the following theorem.

Theorem 3.11. (Wegener, 2000, Theorem 3.2.2). *Let f be a function on variables $X = \{x_1, \ldots, x_n\}$ and let $<$ be a linear order on X. Then the minimal-size OBDD representation of f respecting order $<$ contains as many x_i-nodes as there are different subfunctions $|f_{|\{x_j | x_j < x_i\}}|$.*

Now Theorem 3.11 together with Corollary 3.10 imply the desired result.

Proposition 3.12. $\mathbf{OBDD}_< < \mathbf{SL}$

Proof. Let $f(x_1, \ldots, x_n)$ be a k-switch function and $<$ some linear order of the variables. By Theorem 3.11 and Corollary 3.10 a minimum size OBDD respecting $<$ contains at most $(ik + 1)$ nodes on branching level $i + 1$ for $0 \leq i \leq n - 1$. Therefore such an OBDD has at most $\sum_{i=0}^{n-1}(ik + 1) = \frac{1}{2}kn(n - 1) + n$ nodes which is polynomial in the size of the input switch-list for f of size kn. This proves $\mathbf{OBDD}_< \leq \mathbf{SL}$.

On the other hand, it is easy to see that $\mathbf{SL} \not\leq \mathbf{OBDD}_<$. A good example is the parity function on n variables which is symmetric, and thus changing the order imposed on the set of propositional variables changes neither the minimum size OBDD nor the minimum size SLR. The parity function is well known to have an OBDD of linear size in n, while the number of switches in any SLR is exponential in n. The last fact follows e.g. from an easy observation that every second vector in the truth table that corresponds to an even number changes its parity when the last bit is flipped from 0 to 1 to get the next odd number. Thus every vector which corresponds to an odd number is a switch, and therefore the parity function has $\Omega(2^n)$ switches. □

3.5 Relations among SL (or SL$_<$), PI, and IP Languages

Proposition 3.13. SL *is incomparable with both* **PI** *and* **IP**

Proof. First we prove $\mathbf{SL} \not\leq \mathbf{IP}$ and $\mathbf{SL} \not\leq \mathbf{PI}$. Let us proceed by contradiction assuming $\mathbf{SL} \leq \mathbf{IP}$ ($\mathbf{SL} \leq \mathbf{PI}$ respectively). This assumption together with the relation $\mathbf{OBDD}_< \leq \mathbf{SL}$ (proved in Proposition 3.12) would imply $\mathbf{OBDD}_< \leq \mathbf{IP}$ ($\mathbf{OBDD}_< \leq \mathbf{PI}$ respectively) using the transitivity of succinctness relations. However, both $\mathbf{OBDD}_< \leq \mathbf{IP}$ and $\mathbf{OBDD}_< \leq \mathbf{PI}$ are known to be false, see e.g. (Darwiche & Marquis, 2002) for counterexamples.

Now we shall show $\mathbf{PI} \not\leq \mathbf{SL}$. Let us consider function f on $2n$ variables $x_1, ..., x_n$, $y_1, ..., y_n$ defined as follows. The models of f are the vectors $\{v_i | i = 1, ..., n\}$ where v_i assigns only variables x_i and y_i to 1, and all other variables to 0. Thus f has exactly n models, and the size of its SLR is $O(n^2)$ ($2n$ switches each of size $2n$). Now for an arbitrary subset of indices $S \subseteq 1, ..., n$, let us define a clause $C_S = (\bigvee_{i \in S} x_i \vee \bigvee_{i \notin S} y_i)$. We shall show that for every S, C_S is a prime implicate of f, and therefore f has at least 2^n prime implicates, showing the claim.

Take an arbitrary model v_i of f which by definition satisfies both x_i and y_i. No matter how S was selected, either x_i or y_i appears in C_S satisfying it. Thus C_S is an implicate of f. Now take an arbitrary proper subclause C of C_S. By the definition of C there is an index j such that neither x_j nor y_j appear in C. That means that the model v_j of f falsifies C, which implies that C is not an implicate of f. Thus C_S is prime.

It remains to show $\mathbf{IP} \not\leq \mathbf{SL}$. This relation is more or less a consequence of $\mathbf{PI} \not\leq \mathbf{SL}$ due to the duality between CNFs and prime implicates on one hand and DNFs and prime implicants on the other hand. Consider the negation of the function from the previous proof. Obviously, the SLR of $\neg f$ has exactly the same size as the SLR of f. Vectors $\{v_i | i = 1, ..., n\}$ are now the only non-models of $\neg f$ and for every $S \subseteq 1, ..., n$, we can show that the term $T_S = (\bigwedge_{i \in S} \neg x_i \wedge \bigwedge_{i \notin S} \neg y_i)$ is a prime implicant of $\neg f$. So $\neg f$ has at least 2^n prime implicants, showing the claim.

Take an arbitrary non-model v_i of $\neg f$ which by definition falsifies both $\neg x_i$ and $\neg y_i$. No matter how S was selected, either $\neg x_i$ or $\neg y_i$ appears in T_S falsifying it. Thus T_S is an implicant of $\neg f$. Now take an arbitrary proper subterm T of T_S. By the definition of T there is an index j such that neither $\neg x_j$ nor $\neg y_j$ appear in T. That means that the non-model v_j of $\neg f$ satisfies T, which implies that T is not an implicant of $\neg f$. Thus T_S is prime. □

Since renumbering variables has no effect on the number of prime implicates or prime implicants, we get the same result as above also for the language $\mathbf{SL}_<$.

Corollary 3.14. $\mathbf{SL}_<$ *is incomparable with both* **PI** *and* **IP**

Note, that function f from the proof of Proposition 3.13 has an exponential number of prime implicates with respect to the number of models. This, together with an obvious fact that **MODS** $\not\leq$ **PI**, gives a short proof of the long-standing open problem from (Darwiche & Marquis, 2002, stated as a question mark in Table 3 on page 237).

Corollary 3.15. **PI** $\not\leq$ **MODS** *and hence* **PI** *is incomparable with* **MODS**

It should be noted that the relation **PI** $\not\leq$ **MODS** was first proved in the master thesis (Kaleyski, 2016). However, the construction in (Kaleyski, 2016) is much more complicated and a weaker separation is achieved, namely a quasi-polynomial separation, while the simple construction provided above gives an exponential separation between the number of models and the number of prime implicates.

4. Compilation from SL to OBDD

In this section we shall show that the existential proof of Proposition 3.12 can be extended into a compilation algorithm, which for an input SLR of size kn outputs an OBDD of size $O(kn^2)$. The compilation algorithm works in two steps. First, it compiles the input SLR of function f into a binary decision tree (BDT) which respects the same order of variables as the input SLR. In the second step, it uses the constructed BDT to create an OBDD of f which respects a given prescribed order of variables that may differ from the order used by the input SLR. The output OBDD is constructed level by level and the number of nodes on each level is bounded using Corollary 3.10. Note, that if the input order and the prescribed output order are the same, then the first step suffices, and the constructed linear size BDT gives the output OBDD simply by unifying all zero terminals into a single zero terminal and the same for one terminals.

Our algorithm is similar to the approach taken in (Wegener, 2000, the proof of Theorem 5.7.10). However, our implementation uses techniques that allow us to speed up the minimization process at each level of the constructed OBDD in which we merge nodes representing the same subfunction. To achieve this, we construct "prefix contracted" binary decision trees, which are then used to cache different subfunctions on a given level. This improves the worst-case time $O(|H||G| \log |H|)$ proved by (Savický & Wegener, 1997) and matches the average case time of $O(|H||G|)$ (Tani & Imai, 1994). Our technique accomplishes a worst-case time complexity $O(|H||G|)$, where $|G|$ is the size of the input SLR (which is the same as the size of the BDT constructed in the first step of our algorithm) and $|H|$ is the size of the target OBDD representation.

4.1 Compilation from SL to BDT

Let us consider a SLR of a k-switch function f and construct an equivalent decision tree representation with respect to the same order of variables from the SLR of f. The idea behind the construction is quite simple. Let the order of variables in the input SLR be $x_1, x_2, \ldots, x_n$. Consider the complete binary decision tree of f which branches in the prescribed order, i.e. a tree with n levels of decision nodes and 2^n function values on level $n+1$. Then start a bottom-up process of eliminating redundant decision nodes. In this process, every decision node with both outgoing edges leading to the same function value t is deleted and replaced by an edge from its parent node to function value t. This process obviously stops with a binary decision tree that represents f. Note that this output tree is unique, it depends only on function f and does not depend on the order in which nodes are contracted. What is the size of this unique contracted decision tree? Consider the leaf nodes, that is decision nodes with both outgoing edges going to terminals (function values). Obviously, for every leaf node these two edges necessarily go to different function values (otherwise the node would have been eliminated) and so the path from the root to any leaf node encodes a prefix of some switch of f. Moreover, by the definition of a leaf node, no two leaf nodes can encode a prefix of the same switch, so the number of leaf nodes is upper bounded by the number of switches. It follows that the number of decision nodes in the constructed decision tree is at most n times the number of leaf nodes, which is at most n times the number of switches, which is exactly the size of the input SLR (each switch is a vector of length n). Therefore the constructed decision tree has a linear size with respect to the size of the input SLR.

The above considerations suffice for a proof of an existence of a linear size decision tree equivalent to the input SLR. However, if we want to obtain also a polynomial time compilation procedure that constructs the output decision tree, we have to avoid building the exponentially large initial decision tree. This can be easily avoided by building the output decision tree from top to bottom rather than bottom-up. We start by creating the root node, assigning the interval $[0, 2^n - 1]$ and variable x_1 to the root node, and inserting the root node into a queue. Then we start processing the nodes from the queue in the following manner. Extract the first node v with an assigned interval $[a, b]$ and a variable x_k from the queue. Scan the input switch-list until one of the following two situations occurs:

1. (Non-constant interval) Switch x in the switch-list is found, such that, if interpreted as a binary number, $a < x \leq b$ holds. In this case construct two children nodes v_L, v_R of v, assign variable x_{k+1} to both v_L and v_R, and assign interval $[a, (a+b+1)/2 - 1]$ (the left half of $[a, b]$) to v_L and interval $[(a+b+1)/2, b]$ (the right half of $[a, b]$) to v_R (a is always even, b is always odd, and the length of $[a, b]$ is always a power of 2, so there are no issues with rounding). Insert v_L and v_R to the end of the queue.

2. (Constant interval) Two consecutive switches x, y in the switch-list are found, such that, if interpreted as binary numbers, $x \leq a$ and $b < y$ hold. This means that there is no switch in the interval $[a+1, b]$, and hence all vectors in the interval $[a, b]$ share

the function value of x. So node v can be deleted from the tree of decision nodes and replaced by an edge from the parent node of v to a terminal with function value $f(x)$.

The procedure stops when the queue is empty and constructs exactly the same unique decision tree as the bottom-up procedure described above. The work per decision node is linear in the size of the input switch-list (we scan the switch-list once per node), so the overall complexity of the compilation procedure is at most quadratic in the size of the input switch-list[2].

4.2 Compilation from BDT to OBDD

Now we shall show how to compile a BDT of f which respects the identical order of variables $x_1, \ldots, x_n$ into a polynomial size OBDD of f which respects some other prescribed order of variables $y_1, \ldots, y_n$, where $y_j = x_{\sigma(i)}$ for a given permutation σ. Let us start by defining a special type of binary decision tree.

Definition 4.1. *Let T be a binary decision tree on variables $x_1, \ldots, x_n$ which branches on the variables (on every branch) in this prescribed order. Then T is called a* ***prefix*** *BDT if every branch in T of length l contains the first l decision variables $x_1, \ldots, x_l$, and T is called a* ***contracted*** *BDT if for every decision node x the subtree of T rooted at x contains both terminals 0 and 1.*

Note that the BDT constructed from the input SLR of function f as described in Section 4.1 is a contracted prefix BDT representing f. It is also an easy observation that given a function h on variables $z_1, \ldots, z_k$ it has a one-to-one correspondence with its contracted prefix BDT which respects the given order of variables. We have already observed in Section 4.1 that given h and a fixed order of variables, the resulting contracted prefix BDT is unique. The reverse direction is trivial, given a contracted prefix BDT it of course represents exactly one function h.

The principal idea behind our algorithm is to build a minimum size OBDD of f (which respects the order of variables $y_1, \ldots, y_n$) level by level in a BFS manner, where each node u on a level i of the constructed OBDD will be associated with a contracted prefix BDT representing the corresponding subfunction f_u of f in variables $y_i, \ldots, y_n$ defined by $f_u(y_i, \ldots, y_n) = f(u_1, \ldots, u_{i-1}, y_i, \ldots, y_n)$, where the vector $(u_1, \ldots, u_{i-1}) \in \{0,1\}^{i-1}$ represents the path from the root of the OBDD to the node u. The uniqueness of the contracted prefix BDT representations will allow us to efficiently detect whether two subfunctions on the same level of the OBDD are logically equivalent, which is necessary to build a minimum size OBDD of f.

We can encode a contracted prefix BDT T representing function h into a string on an alphabet $\Sigma = \{0, 1, \ell, r, b\}$ using a DFS traversal of T which writes ℓ when traversing from a parent to its left descendant (we assume that DFS first branches left, i.e. on value 0, at

2. In fact, since the tree is built in a BFS manner level by level, the procedure can be modified to restart the scan of the switch-list from the beginning only once per level, improving the complexity upper bound to n times the size of the input switch-list. Using a smarter data structures which for each decision node define not only the relevant interval of binary numbers but also the relevant interval in the switch-list, the overall complexity can be brought further down to linear time complexity by eliminating by the factor n.

every decision node of T), writes r when traversing from a parent to its right descendant, writes b when backtracking from a decision node, and writes 0 or 1 when traversing from a terminal with that function value. This procedure yields a string of length $O(|T|)$ which is of course also unique for h. Therefore we can check the equivalence of two functions h_1 and h_2 (defined on the same set of variables) simply by comparing the encodings of their contracted prefix BDTs which both respect the same prescribed order of variables. A reasonable data structure to support such string comparisons is a trie. Recall, that given a trie which represents a set R of strings and a string s of length t, one can test in $O(t)$ time whether $s \in R$, in the positive case output the node of the trie that represents s, and in the negative case update the trie by inserting s into R. This gives us the following observation.

Observation 4.2. *Let R be a set of encodings of contracted prefix BDTs stored in a trie and let T be a contracted prefix BDT. Then encoding T into a string s and checking if s is in the trie storing R can be done in $O(|T|)$ time. Moreover, if s is not in the trie storing R, we can add it to the trie in time $O(|T|)$.*

Our construction of the output OBDD will start with a root node (the only node on level 1) and the associated contracted prefix BDT in variables $y_1, \dots, y_n$ which respects the order $x_1, \dots, x_n$, constructed from the input SLR as described in Section 4.1. During the processing of nodes on level i (of the OBDD that is being built), the algorithm keeps a trie $\mathcal{S}$ containing encodings of contracted prefix BDTs associated with nodes on level $i+1$ (starting with an empty $\mathcal{S}$ before the first node on level i is processed).

In a step processing node u on level i with an associated contracted prefix BDT T_u, let $\mathcal{V}$ denote the (possibly empty) set of all already created nodes on level $i+1$, let $\mathcal{T}$ denote the set of all contracted prefix BDTs associated with nodes in $\mathcal{V}$, and let $\mathcal{S}$ denote the trie which contains encodings of all BDTs from $\mathcal{T}$. Now consider the assignment $y_i = 0$. It is easy to modify T_u representing $f(u_1, \dots, u_{i-1}, y_i, \dots, y_n)$ into BDT T_u^0 representing $f(u_1, \dots, u_{i-1}, 0, y_{i+1} \dots, y_n)$. Note that T_u respects the original variable order given by $x_1, \dots, x_n$. If T_u branches on y_i in its root, then T_u^0 is simply the root subtree of T_u corresponding to $y_i = 0$, otherwise T_u^0 originates from T_u by connecting the parent of every node v that branches on y_i directly to the child of v which corresponds $y_i = 0$ (and deleting v). Note that T_u^0 is a prefix BDT, on the other hand it is not necessarily contracted. However, we can transform T_u^0 into a contracted prefix tree by a single DFS pass over T_u^0. When DFS gets to a node v for which both descendants are terminals with the same value c, it connects a parent of v to a terminal c.

After building the contracted prefix BDT T_u^0 we can check if T_u^0 is equivalent to some $T_v \in \mathcal{T}$ using Observation 4.2. If we find such T_v associated with a node v, we connect the branch corresponding to $y_i = 0$ at node u to node v. If such T_v does not exist we create a new node w with an associated contracted prefix BDT $T_w = T_u^0$ and connect the branch corresponding to $y_i = 0$ at a node u to a node w. Moreover, we add w into $\mathcal{V}$, add T_w into $\mathcal{T}$, insert the encoding of T_w into $\mathcal{S}$, and associate the corresponding node in the trie $\mathcal{S}$ with w (so that we have a constant time access to w next time the encoding of T_w is found in $\mathcal{S}$). The procedure for $y_i = 1$ is symmetric.

Assuming that the input k-switch function f is given by a switch-list of size kn, the constructed OBDD has size $O(kn^2)$, and its construction takes $O(k^2n^3)$ time. The complexity is bound by the fact that for each of the $O(kn^2)$ nodes in the OBDD the associated contracted prefix BDT has size $O(kn)$, and all the above-described procedure does when processing a given node of the OBDD is linear in the size of the associated BDT.

Remark 4.3. *The fact that the existential proof of Proposition 3.12 can be extended into a compilation algorithm can be shown also in another way. In Section 6 we show that conditioning (for any variable) can be done in linear time on an input SLR, and so the output OBDD can be efficiently built level by level using subsequent conditioning on variables in the order prescribed for the output. The complexity bottleneck is the same as for the approach described above in this section, namely checking whether a node corresponding to a given subfunction already exists on the given level of the constructed OBDD (in that case it suffices to add an arc to such a node) or not (in which case a new node must be created). If the SLRs for nodes on the current level (the one being built) are cached in an intelligent way to allow such equivalence checks, the overall complexity of the compilation algorithm can be also bounded by $O(k^2n^3)$. However, it seems to us that the two-step algorithm described above is easier to implement.*

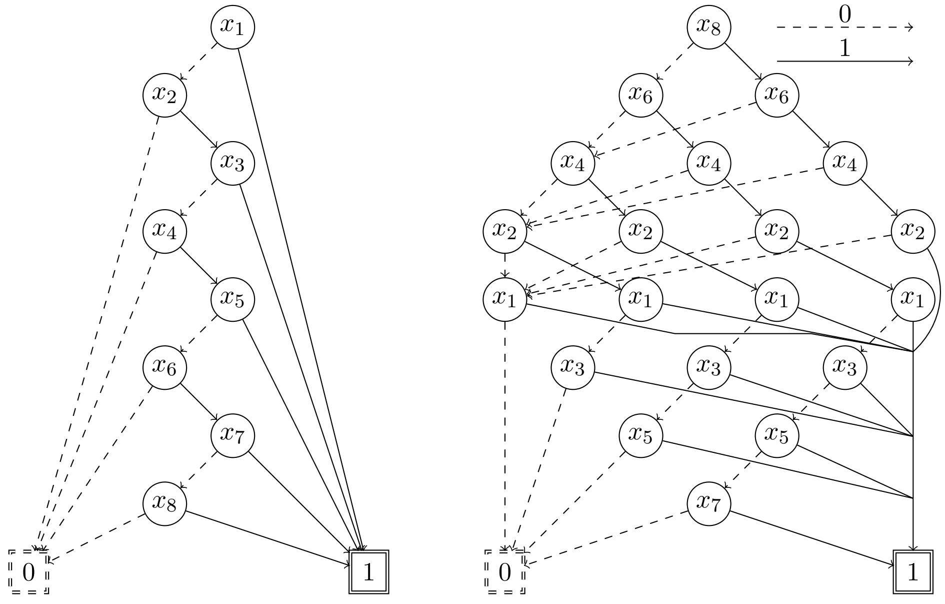
5. Lower Bound for the Size of Target OBDD

In this section, we shall show that the quadratic blowup in the number of nodes in the compilation algorithm from Section 4 which produces an output OBDD of size $O(kn^2)$ is unavoidable. We shall consider a case with $k = 1$ when the input SLR contains a single switch and thus has size $O(n)$, and construct a 1-switch function f for which any OBDD w.r.t. a certain prescribed order of variables has size $\Omega(n^2)$.

Let f be a function on n variables where n is even and the only switch of f with respect to the natural ordering $\pi : x_1 < x_2 < \ldots < x_n$ is $s = (0, 1, 0, 1, 0, 1, \ldots, 0, 1)$. See Figure 3a for an OBDD of f (respecting π) with $n = 8$, which can be produced from the input SLR by the first step of the compilation algorithm from Section 4 (which produces a BDT) and a unification of both terminals. For the output OBDD representation of f we will prescribe the ordering $\sigma : x_n < x_{n-2} < x_{n-4} < \ldots < x_2 < x_1 < x_3 < \ldots < x_{n-3} < x_{n-1}$. See Figure 3b for the minimal output OBDD of f (respecting σ) with $n = 8$. We shall prove that any OBDD representation of f with respect to the ordering σ must have at least i distinct nodes on every level $i \leq n/2$. Thus the total number of nodes on the first $n/2$ levels is at least $\sum_{i=1}^{n/2} i$ which is $\Omega(n^2)$ proving the claim.

Let us proceed by induction on i. Starting the induction for $i = 1$ is trivial, there is a single node on the first level of any OBDD. For $i = 2$ it suffices to verify that $f(x_n = 0) \neq f(x_n = 1)$ which is easy to see from Figure 3a as $f(0, 1, 0, 1, 0, 1, \ldots, 0, 0) = 0$ while $f(0, 1, 0, 1, 0, 1, \ldots, 0, 1) = 1$. Thus, there must be two nodes on the second level of the output OBDD.

For the general induction step let us assume that we have i nodes (where $i \geq 2$) denoted $n_0, \ldots, n_{i-1}$ on the i-th level of an OBDD representing f (this level branches on variable x_{n+2-2i}) and these nodes correspond to pairwise distinct subfunctions $f_0, \ldots, f_{i-1}$ in variables $x_{n+2-2i}, x_{n-2i}, \ldots, x_2, x_1, x_3, \ldots, x_{n-3}, x_{n-1}$. We shall show that there exist $i + 1$ pairwise distinct subfunctions $g_0, \ldots, g_i$ that originate from $f_0, \ldots, f_{i-1}$ by fixing a value of x_{n+2-2i}. This of course implies that there must be at least $i + 1$ distinct nodes $m_0, \ldots m_i$ on level $i + 1$ in any OBDD representing f and respecting the order σ.



(a) Original BDT. (b) Compiled OBDD.

Figure 3: OBDD with different ordering of a function f with variables $x_1, \dots, x_8$ and one switch $(0, 1, 0, 1, 0, 1, 0, 1)$.

First let us consider the case $x_{n+2-2i} = 0$. It is clear from Figure 3a that for any vector $x \in \{0, 1\}^n$ with $x_{n+2-2i} = 0$ (note that $n + 2 - 2i$ is even) when we fix the values of all variables in order π, then the value of $f(x)$ is either decided when setting $x_{n+2-2i} = 0$ or earlier. Hence, $f(x_{n+2-2i} = 0)$ does not depend on any variable x_k for $k > n+2-2i$ and thus in particular on variables $x_n, x_{n-2}, \dots, x_{n+4-2i}$, i.e. on those variables on which the output OBDD respecting order σ branches on the first $i-1$ levels. Therefore fixing the values of variables $x_n, x_{n-2}, \dots, x_{n+4-2i}$ in all possible ways, which is how $f_0, \dots, f_{i-1}$ originate from f, together with fixing $x_{n+2-2i} = 0$ always produces the same function which we denote by g_i. In other words, substituting $x_{n+2-2i} = 0$ into functions $f_0, \dots, f_{i-1}$ produces a single function g_i.

The above observation moreover implies, that every pair of vectors x^j, $x^k \in \{0, 1\}^{n-(i-1)}$ for $0 \leq j < k \leq i - 1$ that guarantees $f_j \neq f_k$ (note that f_j and f_k are functions in $n-(i-1)$ variables with values of $x_{n+4-2i}, \dots, x_{n-2}, x_n$ already fixed) must satisfy $x^j_{n+2-2i} = x^k_{n+2-2i} = 1$. Thus by substituting $x_{n+2-2i} = 1$ into $f_0, \dots, f_{i-1}$ we produce i pairwise distinct functions denoted $g_0, \dots, g_{i-1}$.

It remains to show that g_i is distinct from every function in the set $\{g_0, \dots, g_{i-1}\}$. So let $0 \leq j \leq i - 1$ be arbitrary but fixed, and consider vector $x = (0, 1, 0, 1, \dots 0, p, 1) \in \{0, 1\}^{n+3-2i}$, i.e. we are considering the first $n + 3 - 2i$ variables in order π and p is in position $n+2-2i$. Notice, that all variables on which the output OBDD respecting order σ branches on levels $1, \dots, i-1$ are outside of the scope of indices used by x. Now g_i originates from f_j by setting $p = 0$. Vector x specifies only a partial assignment on variables of g_i, but this partial assignment is sufficient to enforce $g_i(x) = 0$ as can be easily seen from Figure 3a.

	$\neg$**C**	$\wedge$**C***	$\wedge$**BC**	$\wedge$**C**	$\vee$**C***	$\vee$**BC**	$\vee$**C**	**CD**	**SFO**	**FO**
SL$_<$	✓	✓	×	×	✓	×	×	✓	✓	✓
SL	✓	?	?	?	?	?	?	✓	✓	✓

Table 3: Transformations for the **SL** and **SL**$_<$ languages, where ✓means the existence of polytime algorithm and × means that such an algorithm does not exist. Here $\wedge$**C*** and $\vee$**C*** assume that all input SLRs are defined on the same set of variables.

On the other hand, g_j originates from f_j by setting $p = 1$ and the partial assignment x in this case implies $g_j(x) = 1$ which is again easy to see from Figure 3a. Thus g_i is distinct from g_j which finishes the proof of the claim.

6. Transformations

In this section, we investigate which of the common transformations can be implemented to run in polynomial time for languages **SL**$_<$ and **SL**. Precise definitions of the studied transformations can be found in (Darwiche & Marquis, 2002), we give only a short description here:

$\neg$**C** ***Negation*** of a sentence.

$\wedge$**BC** ***Bounded conjunction*** of two sentences.

$\wedge$**C*** ***Conjunction*** of any finite number of sentences on ***the same set of variables***.

$\wedge$**C** ***Conjunction*** of any finite number of sentences.

$\vee$**BC** ***Bounded disjunction*** of two sentences.

$\vee$**C*** ***Disjunction*** of any finite number of sentences on ***the same set of variables***.

$\vee$**C** ***Disjunction*** of any finite number of sentences.

CD ***Conditioning*** of sentence f by term α, i.e. a partial assignment of values forced by satisfying α into f.

SFO ***Singleton forgetting***, i.e. a transformation of f into $\exists x f$ for a variable x.

FO ***Forgetting***, i.e. a transformation of f into $\exists X f$ for is a subset X of variables .

6.1 Negation ($\neg$C)

The negation of SLR L_f representing function f can be produced in constant time by flipping the function value of f at the vector $(0, \ldots, 0)$ while all switches remain the same. This is of course true for both **SL** and **SL**$_<$ languages.

6.2 Conjunctions ($\wedge$BC, $\wedge$C* and $\wedge$C)

Let us consider the conjunction of two SLRs L_f and L_g representing functions f and g. We shall distinguish three cases.

1. ***L_f and L_g respect the same order of variables, and moreover f and g are defined on the same set of variables*** (i.e. the switches in L_f and L_g are vectors of the same length and their coordinates are indexed by variables in the same order). Observe, that if x is neither a switch in L_f nor a switch in L_g, it cannot be a switch in the SLR of $f \wedge g$ which respects the same order of variables. Indeed, if $f(x) = f(x-1) = 0$ or $g(x) = g(x-1) = 0$ then $(f \wedge g)(x) = (f \wedge g)(x-1) = 0$, if $f(x) = f(x-1) = g(x) = g(x-1) = 1$ then $(f \wedge g)(x) = (f \wedge g)(x-1) = 1$. Hence the switch-list of $f \wedge g$ is a subset of the union of the two input switch-lists. Since both input switch-lists are ordered, they can be easily merged into an ordered list and during the merge each switch can be checked whether it is a switch of $f \wedge g$ or not. This can be done in linear time in the size of the input, and moreover this idea can be easily extended to any number of conjuncts and hence to an unbounded conjunction. Note, that to be able to delete duplicate switches and compute the values of $f \wedge g$ efficiently (when checking whether a given x is a switch of $f \wedge g$), it is essential that the input SLRs are ordered. This is one of the reasons why we maintain switch-lists, not just switch sets.

   This special case unfortunately covers only a subset of the **SL**$_<$ language, however, it is a subset that occurs frequently in practical applications (which deal with several different Boolean functions on the same set of variables). It is also important to note, that contrary to SLRs, OBDDs do not support unbounded conjunction even in this restricted case (see Table 7 on page 242 and more details in (Darwiche & Marquis, 2002, pp. 259-261)).

2. ***L_f and L_g respect the same order of variables but do not use the same set of variables***. In this case, the SLR of $f \wedge g$ that respects the same order of variables as L_f and L_g may be exponentially large with respect to the size of L_f and L_g. This may be true even if one of the sets of variables is a strict subset of the other. Consider $f(x_1, \dots, x_n) = \bigvee_{i=1}^{n} \neg x_i$ (the all-one vector is the only false point of f) and $g(x_n) = x_n$. Clearly both L_f and L_g have just one switch with respect to the prescribed order of variables, however, function $f \wedge g$ in variables $x_1, \dots, x_n$, has a switch-list of size $2^n - 1$, because every assignment $x_1, \dots, x_n$, associated with an odd number except of the all-one vector is a model and every assignment associated with an even number is a non-model of $f \wedge g$. Thus the **SL**$_<$ language does not in this case support bounded conjunction in polynomial time.

3. ***L_f and L_g do not respect the same order of variables***. This case is open. We conjecture that the **SL** language does not support bounded conjunction in polynomial time, however, constructing examples where the conjunction is exponentially large with respect to all permutations of variables seems to be difficult.

6.3 Disjunctions ($\vee$BC, $\vee$C* and $\vee$C)

The complexity status for disjunctions is the same as for conjunctions thanks to the constant time negation.

6.4 Conditioning (CD)

Let f be a function on variables $x_1, \ldots, x_n$ and let x_i be an arbitrary variable. We interpret an assignment of variables $x_1, \ldots, x_{i-1}$ as a binary number ℓ, $0 \leq \ell \leq 2^{i-1} - 1$, and denote the corresponding block of consecutive vectors sharing the same prefix ℓ in the truth table of f as B_ℓ. Furthermore, we split B_ℓ into B_ℓ^0 and B_ℓ^1 depending on the value of x_i. We shall show how the SLR of $f_1 = f_{|x_i=1}$ can be obtained from the SLR of f by a single pass through the input switch-list (the process for $f_0 = f_{|x_i=0}$ is similar). We will write the vectors from the truth table of f as triples $(\ell, *, q)$ where $* \in \{0, 1\}$ represents the value of x_i and $\mathbf{0} \leq q \leq 2^{n-i} - 1 = \mathbf{1}$ is a binary number representing $x_{i+1}, \ldots, x_n$ (note that in a vector notation $\mathbf{0}$ is a shorthand for an all zero vector of length $n - i$ and $\mathbf{1}$ is a shorthand for an all one vector of length $n - i$). Similarly, we will write the vectors from the truth table of f_1 as pairs (ℓ, q).

When processing a switch of the type $(\ell, 0, q)$ we just count the parity p of the number of switches having the same prefix $(\ell, 0)$, i.e. the parity of the number of switches in the block B_ℓ^0. After we pass the last switch in B_ℓ^0, let us inspect the next switch in the list. If it differs from $s = (\ell, 1, \mathbf{0})$ (the first vector in B_ℓ^1), p is odd, and $\ell > 0$, we output $s' = (\ell, \mathbf{0})$. In this case s' which originates from s by removing the x_i coordinate becomes a switch of f_1, replacing the odd number of switches in B_ℓ^0. This is because $f_1(s') = f(s)$ differs from $f_1(\ell - 1, \mathbf{1}) = f(\ell - 1, 1, \mathbf{1})$. Note that $(\ell - 1, 1, \mathbf{1})$ is the last vector in $B_{\ell-1}^1$ and so $(\ell - 1, \mathbf{1})$ is a predecessor of s' in the truth table of f_1. If p is even or $\ell = 0$ then the switches in B_ℓ^0 "disappear" without creating any switch for f_1.

When processing a switch of the type $s = (\ell, 1, q)$ where $q > 0$ we simply output $s' = (\ell, q)$ (all such switches of course "survive" the conditioning $x_i = 1$). If $s = (\ell, 1, \mathbf{0})$ (switch s is the first vector in B_ℓ^1) we output $s' = (\ell, \mathbf{0})$ only if p obtained from B_ℓ^0 is even (this includes the case if there are no switches in B_ℓ^0) and $\ell > 0$. Clearly, also in this case s' is indeed a switch of f_1 because its function value differs from the last vector in $B_{\ell-1}^1$ which becomes its predecessor in the truth table of f_1. On the other hand, if p is odd or $\ell = 0$ then s "disappears" without creating a switch for f_1.

If the input SLR has k switches, the above-described process of conditioning on x_i takes $O(n)$ time per switch (and each switch is processed exactly once) and therefore can be implemented to run in $O(kn)$ time. Since the output SLR has at most as many switches as the input SLR, we can repeat the process $|S|$ times to achieve conditioning on any set S of variables in $O(kn^2)$ time. However, the $O(kn)$ time complexity can be maintained even in this case. If we divide the truth table of f into blocks with respect to the least significant variable in S (the rightmost one in the truth table), then instead of the alternating pattern of disappearing and surviving blocks for $|S| = 1$ (as described above) we get a pattern of possibly many disappearing blocks followed by a single surviving block. However, the idea of conditioning can remain the same. We count the parity of the number of switches in between two surviving blocks, treat the first vector in the next surviving block accordingly, then output the remaining switches in the surviving block.

6.5 Forgetting (SFO and FO)

Let f be a function on variables $x_1, \ldots, x_n$ and let x_i be an arbitrary variable. We shall show how the SLR of $f_i = \exists x_i f$ can be obtained from the SLR of f in polynomial time. The procedure can be implemented directly on SLRs, but we find it more understandable if explained on interval representations (IRs) which actually motivated the introduction of SLRs. We first compile the SLR of f into an IR of f, then transform this into an IR of f_i, and finally compile back into an SLR of f_i. The first and third steps take linear time, so it remains to describe the second step. We again (as for **CD**) consider the block structure $B_\ell = B_\ell^0 \cup B_\ell^1$ for $0 \leq \ell \leq 2^{i-1} - 1$ of the truth table of f and also use the $(\ell, *, q)$ notation for the vectors from the truth table of f and (ℓ, q) for the vectors from the truth table of f_i.

When passing through the ordered list of intervals in the IR of f, an interval $[a, b]$ is processed depending on whether $a \in B_\ell^0$ or $a \in B_\ell^1$ (for some ℓ) as follows. For $a = (\ell, 0, q)$ if

(a) $b = (\ell, 0, r)$ (i.e. $[a, b]$ is entirely inside the block B_ℓ^0) then output $[(\ell, q), (\ell, r)]$,

(b) $b = (\ell, 1, r)$ (i.e. $[a, b]$ is entirely inside the block B_ℓ but it spans from B_ℓ^0 to B_ℓ^1) then output $[(\ell, \mathbf{0}), (\ell, r)]$ and $[(\ell, q), (\ell, \mathbf{1})]$,

(c) $b = (k, 0, r)$ for $k > \ell$ (i.e. $[a, b]$ spans from B_ℓ^0 to B_k^0) then output $[(\ell, \mathbf{0}), (k, r)]$,

(d) $b = (k, 1, r)$ for $k > \ell$ (i.e. $[a, b]$ spans from B_ℓ^0 to B_k^1) then output $[(\ell, \mathbf{0}), (k, \mathbf{1})]$.

On the other hand, for $a = (\ell, 1, q)$ if

(e) $b = (\ell, 1, r)$ (i.e. $[a, b]$ is entirely inside the block B_ℓ^1) then output $[(\ell, q), (\ell, r)]$,

(f) $b = (k, 0, r)$ for $k > \ell$ (i.e. $[a, b]$ spans from B_ℓ^1 to B_k^0) then output $[(\ell, q), (k, r)]$,

(g) $b = (k, 1, r)$ for $k > \ell$ (i.e. $[a, b]$ spans from B_ℓ^1 to B_k^1) then output $[(\ell, q), (k, \mathbf{1})]$.

It is easy to check in each of the above seven cases that the models of f in the interval $[a, b]$ really translate to models of f_i in the specified output intervals. However, the output intervals may of course overlap (or even be identical, e.g. a pair of intervals obtained from (a) and (e) may be identical) so another "consolidation" pass through the output is necessary, which replaces any set of overlapping intervals with their union.

The above considerations imply that forgetting a single variable **SFO** can be performed in polynomial time. To see that the same is true for **FO**, we must analyze more carefully case (b), which is the only one when a single interval $[a, b]$ of f may produce two intervals of f_i. If it does, we will call $[a, b]$ a *splitting* interval. Note that if $q \leq r$, the two output intervals merge in the consolidation pass, so $[a, b] = [(\ell, 0, q), (\ell, 1, r)]$ is splitting if and only if $q > r$. Hence $q \neq \mathbf{0}$ and $r \neq \mathbf{1}$ are necessary conditions for $[a, b] = [(\ell, 0, q), (\ell, 1, r)]$ to be a splitting interval.

Observe also, that for $a = (\ell, *, \mathbf{0})$ the interval $[a, b]$ produces only such intervals $[a', b']$ where $a' = (\ell, \mathbf{0})$, and for $b = (k, *, \mathbf{1})$ the interval $[a, b]$ produces only such intervals $[a', b']$ where $b' = (k, \mathbf{1})$.

Putting the facts from the previous two paragraphs together implies, that if we forget the variables in the decreasing order of significance (most significant variables first), neither of the two intervals generated by a splitting interval $[a, b]$ can become a splitting interval when forgetting subsequent variables. Indeed, either the suffix of the left margin of the generated interval is all zeros (and stays all zeros from then on in subsequent forgetting), or the suffix of the right margin of the generated interval is all ones (and stays all ones). Thus, forgetting any subset of variables may altogether at most double the number of intervals on the output, which implies that **FO** can be done in polytime by repeating **SFO**.

7. Queries

In this section, we investigate which of the common queries can be answered in polynomial time for languages $\mathbf{SL}_<$ and **SL**. As for transformations, precise definitions of the studied queries can be found in (Darwiche & Marquis, 2002), we again give only a short description here:

CO ***Consistency*** - test whether a sentence has a model

VA ***Validity*** - test whether a sentence S is a tautology (all 2^n vectors are models of S)

IM ***Implicant Check*** - for a sentence S and a given term T test if $T \models S$

CE ***Clausal Entailment*** - for a sentence S a given clause C test if $S \models C$

SE ***Sentential Entailment*** - for two given sentences S, S' test if $S \models S'$

EQ ***Equivalence*** - for two given sentences S, S' test if $S \equiv S'$

CT ***Model Counting*** - output the number of models of a sentence

ME ***Model Enumeration*** - output all models of a sentence

Since all of the above queries can be answered in polynomial time for the $\mathbf{OBDD}_<$ language (see Table 1 or (Darwiche & Marquis, 2002, Table 5)), the same is true for languages $\mathbf{SL}_<$ and **SL** which can be both compiled into $\mathbf{OBDD}_<$ in polynomial time. It is clear, however, that for most queries direct algorithms using SLRs are much more efficient than indirect algorithms that first compile SLR into an OBDD (which takes $O(k^2n^3)$ time for SLR with k switches on n variables) and only then answer the query. Obviously, some queries are completely trivial for SLRs (consistency, validity) and some can be easily implemented using polynomial time conditioning (clausal entailment, implicant check). It should be also noted that SLRs are very well suited for model counting — a linear number of arithmetic operations (subtractions) on n-bit numbers suffices, and for model enumeration. However, for (the general case of) sentential entailment and equivalence check we currently have no direct algorithms, and so the indirect approach using a compilation to OBDD is the only one we can use now.

In the rest of this section we will describe a polynomial time algorithm for each of the considered queries and determine its time complexity (see Table 4 for a summary). Let us assume that the input is a SLR consisting of k switches representing function f on n variables (i.e. the input SLR has size $O(nk)$).

	CO	**VA**	**IM**	**CE**	**SE***	**SE**	**EQ**	**CT**	**ME**
$\mathbf{SL}_<$	$O(1)$	$O(1)$	$O(kn)$	$O(kn)$	$O(kn)$	$O(k^2n^2)$	$O(kn)$	$O(kn)$	$O(mn)$
SL	$O(1)$	$O(1)$	$O(kn)$	$O(kn)$	$O(k^2n^3)$	$O(k^2n^3)$	$O(k^2n^3)$	$O(kn)$	$O(mn)$

Table 4: Time complexity of queries for the $\mathbf{SL}_<$ and **SL** languages where n is the number of variables, k is the number of switches in the input SLR, and m is the number of models. SE* additionally assumes that both input SLRs are defined on the same set of variables.

7.1 Consistency (CO) and Validity (VA)

To test that f is valid (all assignments are models) it suffices to check $k = 0$ and $f(0, \dots, 0) = 1$. This takes $O(1)$ time. To test that f is consistent (has a model) it suffices to check $k > 0$ or $f(0, \dots, 0) = 1$. This also takes $O(1)$ time. Of course, checking consistency of a function is the same as checking non-validity of its negation (recall from Section 6 that a negation can be constructed in $O(1)$ time).

7.2 Implicant Check (IM) and Clausal Entailment (CE)

Let T be a term (simple conjunction of literals). We have shown in Section 6 that conditioning on a subset of literals takes $O(kn)$ time. Therefore it suffices to set each literal in T to true and then to check the validity of the SLR resulting from this conditioning in $O(1)$ time to determine whether T is an implicant of f or not.

Checking that clause C is an implicate of f is equivalent to checking whether a term $\neg C$ is an implicant of $\neg f$. Thus, due to the fact that SLR of $\neg f$ can be constructed from the SLR of f in $O(1)$ time, also clausal entailment can be checked in $O(kn)$ time.

7.3 Sentential Entailment (SE)

Let f and g be two Boolean functions and L_f, L_g their SLRs with k_f and k_g switches respectively. Let us denote $k = \max\{k_f, k_g\}$. **SE** amounts to checking whether f implies g. This is the same as checking that $\neg f \vee g$ is valid. Let us distinguish three cases in a similar fashion as we did for conjunctions and disjunctions in Section 6.

1. L_f, L_g have the same order of variables and the same set of variables. In this case, we can check **SE** in $O(kn)$ time using negation, disjunction, and validity check.

2. L_f, L_g have the same order of variables and different sets of variables. In this case, we cannot use the same approach as above because the SLR representation of the disjunction may be exponentially large with respect to the input. Hence we compile L_f, L_g into OBDDs G_f, G_g both respecting the same fixed order of variables as L_f, L_g. This can be done in $O(k^2n^2)$ time and the output is of size $O(kn)$ since the first part of the compilation from Section 4.1 suffices to produce G_f, G_g (no need to change the order of variables). Now we can take a negation of G_f in $O(1)$ time, disjoin it with G_g in $O(|G_f| \times |G_g|) = O(k^2n^2)$ time, reduce the result to a canonic OBDD in $O(k^2n^2)$ time, and check validity (see (Wegener, 2000, Theorem 3.3.6. pp. 56) for the

complexity of these operations with OBDDs, more details are also in (Bryant, 1986; Meinel & Theobald, 1998)).

3. L_f, L_g have different orders of variables. In this case, we again compile into OBDDs but this time we have to change the order of variables for one of the input representations. So let us assume that we produce G_f of size $O(kn)$ in $O(k^2n^2)$ time as above (we keep the order of variables) and G_g of size $O(kn^2)$ in $O(k^2n^3)$ time by the compilation algorithm from Section 4.2 (here we change the order of variables to the one in G_f). Now we proceed as above taking a negation, disjunction and checking validity. In this case, we need $O(|G_f| \times |G_g|) = O(k^2n^3)$ time.

7.4 Equivalence Check (EQ)

For **EQ** the situation is simpler than for **SE** since when testing equivalence we may assume that f and g are defined on the same set of variables. If L_f, L_g have the same order of variables then **EQ** can be obviously tested in $O(kn)$ time by performing two **SE** checks. In fact, in this case, if $f \equiv g$ then L_f and L_g must be identical, as for a given function and a given order of variables its SLR is uniquely defined. So instead of two **SE** checks we may just test in $O(kn)$ whether L_f and L_g are identical, which is easy to do as the switch-lists are ordered.

If the order of variables differs, we proceed similarly as in **SE** by compiling L_f of size $O(kn)$ into G_f of size $O(kn)$ in $O(k^2n^2)$ time and L_g of size $O(kn)$ into G_g of size $O(kn^2)$ in $O(k^2n^3)$ time. Now we can save some time by checking the equivalence of f and g by testing the isomorphism of the reduced forms of G_f and G_g in $O(kn^2)$ time (Wegener, 2000, Theorem 3.3.1. pp. 51), but asymptotically this does not help as the complexity of the compilation step dominates.

7.5 Model Counting (CT) and Model Enumeration (ME)

Let $p_1, \ldots, p_k$ be the switches of f understood as n-bit binary numbers. If $f(1, \ldots, 1) = 1$ then set $p_{k+1} = 2^n$. Now there are two cases. If $f(0, \ldots, 0) = 1$ then set $p_0 = 0$ and the number of models equals to $\sum_{i \in I}(p_i - p_{i-1})$, where I is set of odd indices, if $f(0, \ldots, 0) = 0$ then the number of models equals to $\sum_{i \in I}(p_i - p_{i-1})$, where I is set of even indices. In either case, the computation consists of $O(k)$ addition and subtraction operations on n-bit numbers which takes $O(kn)$ time.

Model enumeration can be performed in a similar manner as above by outputting models between pairs of switches p_{i-1} and p_i for $i \in I$ depending on the values of $f(0, \ldots, 0)$ and $f(1, \ldots, 1)$ as in model counting. The time complexity is of course linear in the size of the output and takes $O(mn)$ time, where m is the number of models.

8. Conclusions

The main aim of this paper is to include the languages $\mathbf{SL}_<$ and **SL** into the Knowledge Compilation Map (Darwiche & Marquis, 2002) and to argue that they may in some situations constitute reasonable target languages for knowledge compilation. This aim is justified by completing three subtasks: (1) derive the relative succinctness of $\mathbf{SL}_<$ and **SL** compared to the languages already considered in (Darwiche & Marquis, 2002), (2) establish the complexity status of common transformations for $\mathbf{SL}_<$ and **SL**, and (3) do the same for common queries. This goal is achieved with few open problems remaining, namely the complexity of conjunctions and disjunctions for **SL**.

The results in this paper are dependent on the fact that vectors in the truth table are assumed to be ordered by the natural lexicographic order. i.e. by standard inequalities when vectors are interpreted as binary numbers. There are other orders which are quite natural and can be generated effectively. For instance, one can order vectors based on the number of ones (and complement this by some natural order on the sets of vectors with the same number of ones). Such an order has quite different properties, note that e.g. the parity function which has an exponentially large SLR with respect to the standard order of vectors has a linear size SLR with respect to this less standard one. Examining the properties of SLRs with respect to non-standard orders of vectors may be the subject of a future study.

Acknowledgements

The authors gratefully acknowledge a support by Czech Science Foundation (Grant 19-19463S), by SVV project number 260 575 and by Charles University project UNCE/SCI/004.

References

Ausiello, G., D'Atri, A., & Sacca, D. (1986). Minimal representation of directed hypergraphs. *SIAM Journal on Computing*, *15*, 418–431.

Boros, E., Čepek, O., Kogan, A., & Kučera, P. (2009). A subclass of Horn CNFs optimally compressible in polynomial time. *Annals of Mathematics and Artificial Intelligence*, *57*, 249–291.

Boros, E., Čepek, O., & Kučera, P. (2013). A decomposition method for CNF minimality proofs. *Theoretical Computer Science*, *510*, 111–126.

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

Čepek, O., & Chromý, M. (2020). Switch-list representations in a knowledge compilation map. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 1651–1657. ijcai.org.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *Journal Of Artificial Intelligence Research*, *17*, 229–264.

Hammer, P. L., & Kogan, A. (1993). Optimal compression of propositional Horn knowledge bases: Complexity and approximation. *Artificial Intelligence*, *64*, 131–145.

Hammer, P. L., & Kogan, A. (1995). Quasi-acyclic propositional Horn knowledge bases: Optimal compression. *IEEE Transactions on Knowledge and Data Engineering*, *7*(5), 751–762.

Huang, C.-Y., & Cheng, K.-T. (1999). Solving constraint satisfiability problem for automatic generation of design verification vectors. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop.*

Kaleyski, N. S. (2016). Boolean methods in knowledge compilation. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics.

Kronus, D., & Čepek, O. (2008). Recognition of positive 2-interval Boolean functions. In *Proceedings of 11th Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty*, pp. 115–122.

Le Berre, D., Marquis, P., Mengel, S., & Wallon, R. (2018). Pseudo-boolean constraints from a knowledge representation perspective. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI18, p. 18911897. AAAI Press.

Lewin, D., Fournier, L., Levinger, M., Roytman, E., & Shurek, G. (1995). Constraint satisfaction for test program generation. In *IEEE 14th Phoenix Conference on Computers and Communications*, pp. 45–48.

Meinel, C., & Theobald, T. (1998). *Algorithms and Data Structures in VLSI Design* (1st edition). Springer-Verlag, Berlin, Heidelberg.

Savický, P., & Wegener, I. (1997). Efficient algorithms for the transformation between different types of binary decision diagrams. *Acta Inf.*, *34*(4), 245–256.

Schieber, B., Geist, D., & Zaks, A. (2005). Computing the minimum DNF representation of boolean functions defined by intervals. *Discrete Applied Mathematics*, *149*, 154–173.

Tani, S., & Imai, H. (1994). A reordering operation for an ordered binary decision diagram and an extended framework for combinatorics of graphs.. In *Proceedings of the 5th International Symposium on Algorithms and Computation.*, Vol. 834, pp. 575–583.

Umans, C. (2001). The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, *63*(4), 597–611.

Umans, C., Villa, T., & Sangiovanni-Vincentelli, A. L. (2006). Complexity of two-level logic minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, *25*(7), 1230–1246.

Čepek, O., & Chromý, M. (2020). Compiling sl representations of boolean functions into obdds. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2020, Fort Lauderdale, Florida, USA, January 6-8.*

Čepek, O., & Hušek, R. (2017). Recognition of tractable dnfs representable by a constant number of intervals. *Discrete Optimization*, *23*, 1–19.

Čepek, O., Kronus, D., & Kučera, P. (2008). Recognition of interval Boolean functions. *Annals of Mathematics and Artificial Intelligence*, *52*(1), 1–24.

Wegener, I. (2000). *Branching Programs and Binary Decision Diagrams: Theory and Applications.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.