Switch lists in the landscape of knowledge representation languages

Ondřej Čepek
Faculty of Mathematics and Physics, Charles University
Malostranske namesti 25
118, 00 Praha 1, Czech Republic

Abstract

A switch-list representation (SLR) of a Boolean function is a compressed truth table representation of a Boolean function in which only the function value of the first row in the truth table and an ordered list of switches are stored. A switch is a Boolean vector whose function value differs from the value of the preceding Boolean vector in the truth table. In this short paper we outline scenarios under which SLRs constitute a better representation language than standard representation languages such as CNFs and OBDDs. Furthermore, we outline a possible approach to constructing a compiler from CNFs to SLRs which is a necessary tool for verifying practical usefulness of SLRs.

Introduction

Knowledge representation languages constitute different formalisms for representing Boolean functions. 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 and has many applications to real life problems. Well known representation languages include various types of Boolean formulas (e.g. CNFs and DNFs), various types of binary decision diagrams (BDDs, FBDDs, OBDDs), and negational normal forms (NNF, DNNF, d-DNNF). A Boolean function can also be represented by a truth table or a list of models.

The task of transforming one of the representations of a given function f into another representation of f (e.g. transforming a DNF into an OBDD or a DNNF into a CNF) is called *knowledge compilation*. A comprehensive review paper on knowledge compilation (Darwiche and Marquis 2002) introduces a Knowledge Compilation Map (KCM). 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.

Copyright © 2021by the authors. All rights reserved.

The paper (Le Berre et al. 2018) included Pseudo-Boolean constraint ($\mathbf{PBC}$) and Cardinality constraint ($\mathbf{CARD}$) languages into KCM by adding them into the succinctness diagram, and by proving the complexity status of almost all queries and transformations introduced in (Darwiche and Marquis 2002). The same was later achieved for languages $\mathbf{SL}$ and $\mathbf{SL}_<$ based on switch list representations in (Čepek and Chromý 2020). In this paper, we discuss several scenarios under which $\mathbf{SL}$ and $\mathbf{SL}_<$ are superior to other known representation languages and suggest a research direction which may show their practical usefulness. Let us start by defining the two languages considered in this paper.

Definition 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 $\mathbf{SL}$ is the union of $\mathbf{SL}_<$ languages over all total orders on the set PS.*

Furthermore, 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.

In the following three sections we compare $\mathbf{SL}$ and $\mathbf{SL}_<$ languages with other standard representation languages with respect to (1) their relative succinctness, (2) the complexity of common transformations, and (3) the complexity of common queries. In the last section we outline a possible approach for compilation into the $\mathbf{SL}$ language.

Relative Succinctness

Two sentences (possibly from two different propositional languages) are called *logically equivalent* if they represent the same Boolean function. Let us now define the notion of relative succinctness.

Definition 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* $\mathbf{K} \not\leq \mathbf{L}$*) then* $\mathbf{L}$ *is* strictly more succinct *than* $\mathbf{K}$*, denoted* $\mathbf{L} < \mathbf{K}$*.*

The diagram in Figure 1 from (Darwiche and Marquis 2002) summarizes the strict succinctness relations of many commonly used propositional languages. It is amended here by the results from (Le Berre et al. 2018) dealing with the $\mathbf{PBC}$ and $\mathbf{CARD}$ languages and the results from (Čepek and Chromý 2020) dealing with $\mathbf{SL}$ and $\mathbf{SL}_<$ languages.

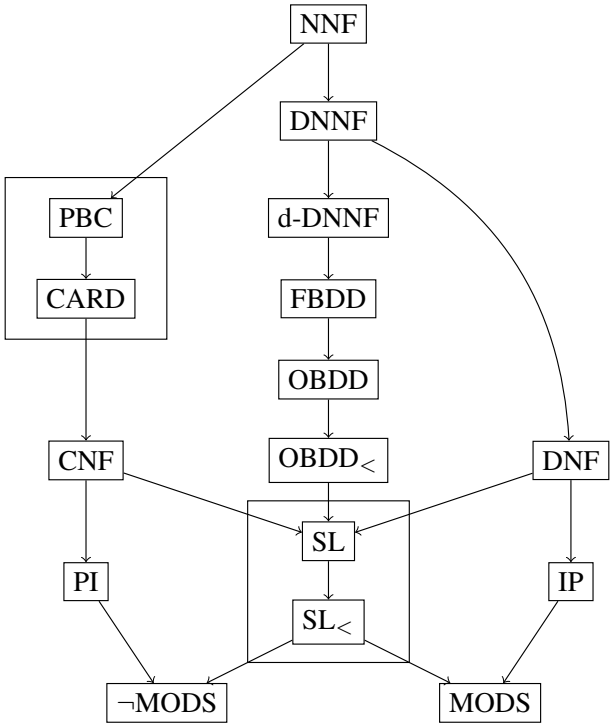

Figure 1: $\mathbf{L} \rightarrow \mathbf{K}$ means $\mathbf{L} < \mathbf{K}$

When arguing about the quality of a representation language $\mathbf{L}$ it is customary to compare its properties to the properties of its neighbours in the succinctness diagram. Obviously, $\mathbf{L}$ should have better properties (support more queries and transformations in polynomial time) than the more succinct languages "above it" and may support fewer queries and transformations than the less succinct languages below it. Thus, when assessing the quality of $\mathbf{SL}$ and $\mathbf{SL}_<$, we should compare them primarily to $\mathbf{CNF}$, $\mathbf{DNF}$, $\mathbf{OBDD}$ and $\mathbf{OBDD}_<$ on one hand and to the language $\mathbf{MODS}$ of all models on the other hand (the language $\neg\mathbf{MODS}$ is not commonly used and is added into the diagram only for symmetry reasons). It is also interesting to compare the properties of $\mathbf{SL}$ and $\mathbf{SL}_<$ to the properties of the language $\mathbf{IP}$ of all prime implicants (and its symmetric language $\mathbf{IP}$ of all prime implicates). The languages $\mathbf{IP}$ and $\mathbf{PI}$ lie on the same level in the succinctness diagram as $\mathbf{SL}$ and $\mathbf{SL}_<$ and are incomparable with them with respect to the succinctness relation.

As we shall see in the next two sections $\mathbf{SL}$ and $\mathbf{SL}_<$ have a wider set of supported queries and transformations than $\mathbf{CNF}$, $\mathbf{DNF}$, $\mathbf{OBDD}$ and $\mathbf{OBDD}_<$ (and even $\mathbf{IP}$ and $\mathbf{PI}$) which may make $\mathbf{SL}$ and $\mathbf{SL}_<$ a good choice as target languages for knowledge compilation under certain scenarios which exploit the supported queries and transformations not supported by the other representation languages.

Queries

Standard queries considered in the Knowledge compilation map (Darwiche and Marquis 2002) are:

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

VA *Validity* - test whether sentence S is a tautology

CE *Clausal Entailment* - test $S \models C$ for S and clause C

IM *Implicant Check* - test $T \models S$ for S and term T

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

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

CT *Model Counting* - output the number of models of S

ME *Model Enumeration* - output all models of S

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

Table 1: ✓means "can be answered in poly-time" and ○ means "cannot be answered in poly-time unless P=NP".

Table 1 summarizes the complexity of all standard queries for languages considered in (Darwiche and Marquis 2002) as well as those added to KCM by (Le Berre et al. 2018) and (Čepek and Chromý 2020) where the order of languages in the table roughly corresponds to their positions in the succinctness diagram in Figure 1. The proofs of all results presented in Table 1 can be found in the three papers cited

above. Notice that $\mathbf{SL}$ and $\mathbf{SL}_<$ allow answering all queries in polynomial time so these languages are just as good for query answering as $\mathbf{MODS}$ despite the fact that for certain Boolean functions the SLR representation (sentence) is exponentially smaller than the list of models. When compared to more succinct languages $\mathbf{SL}$ and $\mathbf{SL}_<$ are much better than $\mathbf{CNF}$ and $\mathbf{DNF}$ (which support very few queries in polynomial time) but also better than $\mathbf{IP}$ and $\mathbf{PI}$ which do not support model counting. This immediately suggests a scenario in which $\mathbf{SL}$ and $\mathbf{SL}_<$ are good target languages.

Scenario 1: The input representation is a DNF or a CNF and we want to answer a number of queries not supported in polynomial time, e.g. implicant checks or clausal entailment checks for a number of terms or clauses. Then it would make sense to first compile the input into a SLR and answer all queries using this SLR. For a large number of queries (large number of different terms or clauses for the same function) this approach may pay off even if the compilation is expensive. However, this approach is not usable if the compilation step leads to an exponential blow-up of the representation, which must happen for some functions due to the strict succinctness relation, but may not happen for many others. To test this approach in practice a compiler from $\mathbf{CNF}$ (or $\mathbf{DNF}$) to $\mathbf{SL}$ which currently does not exist is needed.

The languages based on SLRs are also slightly better than the $\mathbf{OBDD}$ language which does not support sentential entailment if the two input OBDDs respect different orders of variables. The only language considered in (Darwiche and Marquis 2002) with the same set of supported queries is $\mathbf{OBDD}_<$. Hence, the advantage of $\mathbf{SL}$ is that it does not require the same order of variables for all inputs to guarantee polynomial time bounds on all queries. This may be important when we have no control on variable order during the process of generating the representation, or the input representations come from different sources. However, queries alone would not justify the use of SLR based languages instead of OBDD based languages which are almost as good in query answering (sentential entailment check being the only exception) while being strongly more succinct. To see more important advantages of $\mathbf{SL}$ and $\mathbf{SL}_<$ over $\mathbf{OBDD}$ and $\mathbf{OBDD}_<$ we have to look at transformations.

Transformations

Standard transformations considered in the Knowledge compilation map (Darwiche and Marquis 2002) include:

- $\mathbf{CD}$ *Conditioning* of sentence S by term T, i.e. a partial assignment of values forced by satisfying all literals in T.
- $\mathbf{SFO}$ *Singleton forgetting* which transforms S into $\exists x S$ for a variable x.
- $\mathbf{FO}$ *Forgetting* which transforms S into $\exists X S$ for a subset X of variables.
- $\wedge\mathbf{C}$ *Conjunction* of any finite number of sentences.
- $\vee\mathbf{C}$ *Disjunction* of any finite number of sentences.
- $\neg\mathbf{C}$ *Negation* of a sentence.

	CD	FO	SFO	∧C	∨C	¬C
NNF	✓	○	✓	✓	✓	✓
DNNF	✓	✓	✓	○	✓	○
d-DNNF	✓	○	✓	✓	✓	✓
CARD	✓	○	?	✓	•	•
PBC	✓	•	•	✓	•	•
BDD	✓	○	✓	✓	✓	✓
FBDD	✓	•	○	•	•	✓
OBDD	✓	•	✓	•	•	✓
$\text{OBDD}_<$	✓	•	✓	•	•	✓
CNF	✓	○	✓	✓	•	•
DNF	✓	✓	✓	•	✓	•
IP	✓	•	•	•	•	•
SL	✓	✓	✓	•	•	✓
$\text{SL}_<$	✓	✓	✓	•	•	✓
MODS	✓	✓	✓	•	•	•

Table 2: ✓ means "can be done in poly-time", • means "cannot be done in poly-time" and ○ means "cannot be done in poly-time unless P=NP"

Table 2 summarizes the complexity of transformations for languages considered in (Darwiche and Marquis 2002) as well as those added to KCM by (Le Berre et al. 2018) and (Čepek and Chromý 2020). Once again, the order of languages in the table roughly corresponds to their positions in the succinctness diagram in Figure 1. All results for $\mathbf{SL}$ and $\mathbf{SL}_<$ are due to (Čepek and Chromý 2020) except of $\wedge\mathbf{C}$ and $\vee\mathbf{C}$ for $\mathbf{SL}$ which are due to (Mengel 2022). The results for all other languages can be found in (Darwiche and Marquis 2002) and (Le Berre et al. 2018).

If we think of a list of models as a compressed truth table and of a SLR as an even more compressed truth table, it is quite remarkable that $\mathbf{SL}$ actually supports more transformations than $\mathbf{MODS}$, in particular it supports negation (in fact, negation takes constant time on a SLR !!!) while the list of models may grow exponentially when a negation is taken. Given that the set of supported queries is the same, $\mathbf{SL}$ is in all aspects a better language than the strictly less succinct $\mathbf{MODS}$ language.

When we compare $\mathbf{SL}$ (and $\mathbf{SL}_<$) to the languages on the same level of the succinctness diagram, i.e. to $\mathbf{IP}$ (and $\mathbf{PI}$) we see that $\mathbf{SL}$ vastly outperforms $\mathbf{IP}$ as it additionally supports both general and singleton forgetting and negation. A more interesting comparison is with the strictly more succinct languages. When compared to $\mathbf{CNF}$ and $\mathbf{DNF}$ the number of supported queries is almost the same: $\mathbf{SL}$ (and $\mathbf{SL}_<$) additionally support negation while $\mathbf{CNF}$ additionally supports conjunction and $\mathbf{DNF}$ additionally supports disjunction (those two properties are obvious as CNFs and DNFs are "custom made" for those operations). Recall, however, that $\mathbf{SL}$ (and $\mathbf{SL}_<$) significantly outperform $\mathbf{CNF}$ and $\mathbf{DNF}$ in query answering which balances out the fact that they are strictly less succinct.

$\mathbf{SL}$ and $\mathbf{SL}_<$ also support forgetting (the general case, not just singleton forgetting) which distinguishes them from $\mathbf{OBDD}$ and $\mathbf{OBDD}_<$ that do not support general forgetting. An additional advantage is that SLRs also support con-

junction and disjunction under the 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 both $\mathbf{OBDD}$ and $\mathbf{OBDD}_<$ fail to support conjunction and disjunction even in this restricted case which may appear very naturally in practical applications.

Scenario 2: The collection of supported queries and transformations suggests that $\mathbf{SL}$ (and $\mathbf{SL}_<$) may be a good choice for a target compilation language 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 strictly more succinct representation would support such a scenario in polynomial time. Again, an obvious problem for this approach is a lack of compilation algorithms with $\mathbf{SL}$ as the target representation language.

Compilation into the SL language

Given a truth table or a list of models, it is obviously trivial to compile such a representation into a SLR in polynomial time, often getting an exponential compression. This is unfortunately not a very interesting case, as truth tables and lists of models are rarely used in practice because of their size. It follows, that such compilation may be useful only for small data. However, in such a case it makes perfect sense to do this compression step as no polynomial time properties (queries and transformations) will be lost. In fact, just the contrary will happen - recall the discussion about the superiority of $\mathbf{SL}$ to $\mathbf{MODS}$).

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 logically equivalent SLR. Such a compilation may make sense if we want to use the BDT representation for query answering or manipulate the BDT using some standard transformations. This will be possible once a library of query and transformation routines exists (it is now being written).

Let us finally consider the most interesting example of a compilation algorithm with $\mathbf{SL}$ as a target language. Paper (Čepek and Hušek 2017) studies k-switch functions (recall that function f is a k-switch function if there exists a permutation of its variables for which the SLR of f consists of at most k switches) given by DNFs from tractable classes. In this context a class of DNFs is tractable if it admits polynomial time validity check. The main result in (Čepek and Hušek 2017) is 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) and either outputs a k-switch SLR or fails, the latter meaning that no SLR of the input function with at most k switches exists. This result can be easily adapted to a compilation algorithm from tractable CNFs (which admit polynomial time consistency check) to SLRs as follows: take a negation of the input (which transforms a tractable CNF into a tractable DNF), compile it into a SLR using the algorithm from (Čepek and Hušek 2017), and if the compilation step succeeds, take a negation of the output SLR.

This algorithm is of course of a very limited interest in practice because very few DNF (or CNF) encodings of practical problems fall into some known tractable class of DNFs (or CNFS). Nevertheless, it leads to an interesting research question. Can we extend the algorithm to DNF (or preferably CNF) inputs outside of tractable classes by running a SAT solver whenever the algorithm from (Čepek and Hušek 2017) calls a dedicated polynomial time falsifiability algorithm (here we use that the input is tractable)? This should be possible since calling a dedicated polynomial time falsifiability algorithm is equivalent to calling a dedicated polynomial time satisfiability algorithm for tractable CNF inputs. This should allow us to run the compilation algorithm on selected benchmark CNFs, and by gradually increasing the parameter k test whether some benchmark CNFs have reasonably small switch list representations. This would allow us to validate Scenarios 1 and 2 suggested above on real world data and compare the properties of SL representations to the properties of logically equivalent CNFs (especially with respect to query answering).

Another research question independent of the one formulated in the previous paragraph is whether there exist other nontrivial special classes of CNF, DNF, OBDD, or other common representations, which can be efficiently compiled into SLRs by polynomial time compilation algorithms (similarly to the algorithm from (Čepek and Hušek 2017).

Acknowledgements

The authors gratefully acknowledge a support by Czech Science Foundation (Grant 19-19463S) and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

References

Čepek, O., and Chromý, M. 2020. Properties of switch-list representations of boolean functions. *Journal Of Artificial Intelligence Research* 69:501–529.

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

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

Le Berre, D.; Marquis, P.; Mengel, S.; and Wallon, R. 2018. Pseudo-boolean constraints from a knowledge representation perspective. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, 1891–1897. AAAI Press.

Mengel, S. 2022. No efficient disjunction or conjunction of switch-lists. *arxiv.org/abs/2203.04788*.