Puzzle Solving with Proof

Writing a Verified SAT Encoding Chain in HOL4

Master's thesis in Computer Science – Algorithms, Languages, and Logic

SOFIA GILJEGÅRD
JOHAN WENNERBECK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Puzzle Solving with Proof

Writing a Verified SAT Encoding Chain in HOL4

SOFIA GILJEGÅRD
JOHAN WENNERBECK

Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Puzzle Solving with Proof
Writing a Verified SAT Encoding Chain in HOL4
SOFIA GILJEGÅRD
JOHAN WENNERBECK

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Puzzle Solving with Proof
Writing a Verified SAT Encoding Chain in HOL4
SOFIA GILJEGÅRD, JOHAN WENNERBECK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The development of SAT solvers is an ongoing research area, driven by the possibility to use SAT solvers to solve a wide range of problems. This development has made SAT solvers fast, and they solve increasingly more difficult problems. In order to take advantage of this development, the encoding process to SAT is crucial and must be correct. The purpose of this project is to implement a *verified* chain of encoding functions. The encoding functions are written in and verified using HOL4 as part of CakeML. The result is three different versatile datatypes that can be used, together with a SAT solver, to solve problems containing Boolean variables, natural numbers, and unordered sets. The usage of the three datatypes is demonstrated on different logical puzzles. Using the verified SAT encoding chain allows encodings of new problems to conjunctive normal form to be made with a reduced number of proofs needed in the verification process. We believe that this is the first instance of a verified SAT encoding chain in HOL4.

Keywords: encodings, SAT, CNF, formal methods, formal verification, interactive theorem prover, HOL, CakeML, puzzles

Acknowledgements

We would like to thank our supervisor Magnus Myreen for all the support, great ideas and challenging discussions during the thesis. Your guidance and your advice has been invaluable for many breakthroughs. We would also like to thank our examiner Wolfgang Ahrendt for the valuable feedback and your engagement in the formal methods course that made us both interested in the subject. Furthermore, we would like to give our biggest gratitude to the HOL4/CakeML community for all lighting fast responses to our many questions.

Sofia Giljegård, Johan Wennerbeck
Gothenburg, June 2021

Contents

1

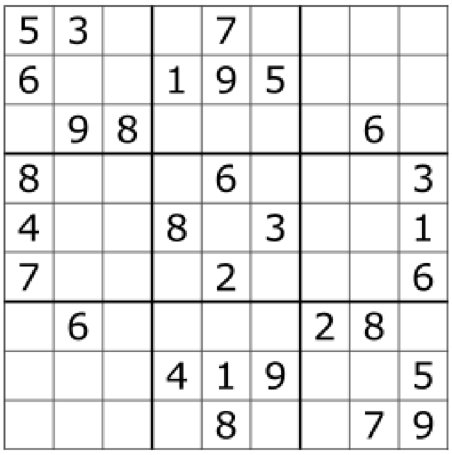
Introduction

Satisfiability testing, in the form of SAT solvers, is an ongoing research area [1, 2]. It is driven by, for example, the need to verify software and hardware [3]. SAT solvers solve Boolean satisfiability (SAT) problems, which involves telling whether a Boolean formula has an assignment of the Boolean variables that makes the formula true. Even though SAT problems are NP-complete, and therefore difficult to solve, SAT solvers are efficient. In order to use a SAT solver to solve a problem, the problem must be encoded to SAT. The encoding process is crucial and must be correct.

Much research has been done on how to encode problems to SAT efficiently. However, not many of these encoding methods have been formally verified. By formally verifying the encoding process, one is confident that the solution to the original problem and the solution to the encoded problem are the same.

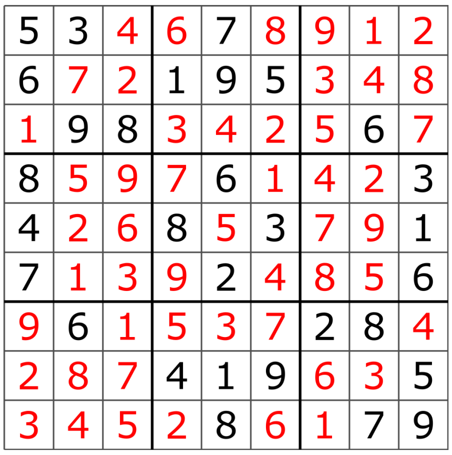
This project aims to implement verified SAT encoding functions for logical puzzles. Say, for example, that we would like to encode a sudoku, such as the one to the left in Figure 1.1. Sudoku is one of the most well-known logical problems, and it consists of a 9×9 grid where each cell should be filled with one of the numbers 1 to 9. It is solved if, within each row, column, and 3 x 3 block, each number appears exactly once. A sudoku is valid if it has exactly one solution.

The rules of a sudoku can easily be expressed as logical constraints. For example, the fact that the first row should contain each number exactly once is one such constraint. SAT solvers are ideal for solving logical constraints. Therefore if we encode the problem to SAT, we can use a SAT solver to find a solution to the sudoku, or to determine if it is valid.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Unsolved



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(b) Solved

Figure 1.1: Example sudoku

The encoding from sudoku to SAT encodes a cell, which is a number between 1 and 9, to nine Boolean variables. Each Boolean variable corresponds to filling that cell with a particular number. The rules of a sudoku are encoded to comparing these Boolean variables to make sure that there exists no duplicate numbers in any row, column, or block.

1.1 Contributions

This project implements *verified* functions for encoding problems to SAT problems. These functions can be used to encode a range of different problems, where sudoku is one example. Chapter 3 explains how the encoding of our sudoku is performed.

The encoding functions are written in and verified using HOL4. They are implemented as part of CakeML, a functional programming language, and tools built around the language. One such tool is a verified compiler, with which our functions are compiled. By compiling the functions with the verified compiler, one is confident that the compiler does not miscompile the given function. Chapter 2 contains relevant information about CakeML, HOL4, and other core concepts for the project.

The result of the project is a collection of datatypes with verified encodings to SAT problems. A description of the datatypes and their encoding functions can be found in Chapter 4, and a description of the verification process can be found in Chapter 5. There are three main datatypes, intended to be used for different logical problems.

The first datatype is intended to use for problems containing only Boolean variables. One such problem could be, for example, the n-queens problem, explained in Section 6.1.1. The goal of the n-queens problem is to place n queens on a $n \times n$ sized chessboard so that no two queens threaten each other.

The second datatype allows for natural numbers variables. These variables have an upper and a lower bound, and equations using them include adding and comparing variables. This datatype can be used to encode logical problems involving adding natural numbers, such as killer sudoku, which is explained in Section 6.2.2. Killer sudoku is a sudoku version with additional constraints involving addition of numbers.

The last datatype is made for use with problems containing variables of other types. One example of this is the graph coloring problem, which is explained in Section 6.1.2. The goal of the graph coloring problem is to color all vertices in a graph so that no connected vertices have the same color. Here the variables are vertices in a graph, and they belong to a set of colors.

The focus has been on proving the encodings correct, not to make them efficient. A discussion of how they compare to other encodings can be found in Chapter 7.

2

Background

This chapter introduces fundamental concepts needed to understand the rest of the report. Section 2.1 gives necessary background information on the SAT problem type. The project will be part of CakeML, which Section 2.2 introduces. CakeML is built upon HOL4. Section 2.3 describes HOL4 in order to give the reader an idea of how the tool is used.

2.1 The SAT Problem Type

The task of a SAT solver is to solve Boolean Satisfiability Problems. This type of problem consists of Boolean variables together with different operators, such as negation or implication.

SAT solvers take their input in a standardized form, called conjunctive normal form (CNF). The definition of CNF is

$$\begin{aligned} \text{literal} &= \text{a boolean variable or the negation of a boolean variable} \\ \text{clause} &= \text{a disjunction of literals} \\ \text{CNF} &= \text{a conjunction of clauses.} \end{aligned}$$

For example,

$$(b_1 \vee \neg b_2) \wedge (b_2),$$

where b_1 and b_2 are Boolean variables, is a formula in CNF consisting of two clauses.

A SAT solver determines whether a formula of this type is satisfiable or not. A formula is satisfiable if there exists an assignment of the Boolean variables that makes the equation evaluate to true [4]. For example, our example above evaluates to true when both b_1 and b_2 evaluate to true.

2.2 CakeML

CakeML is a functional programming language and an ecosystem of proofs and tools built around the language. The base is a compiler, written and proved correct in HOL4 [5]. By writing and proving functions in HOL4 and compiling them with the CakeML compiler, one is confident that the compiler does not miscompile the given function.

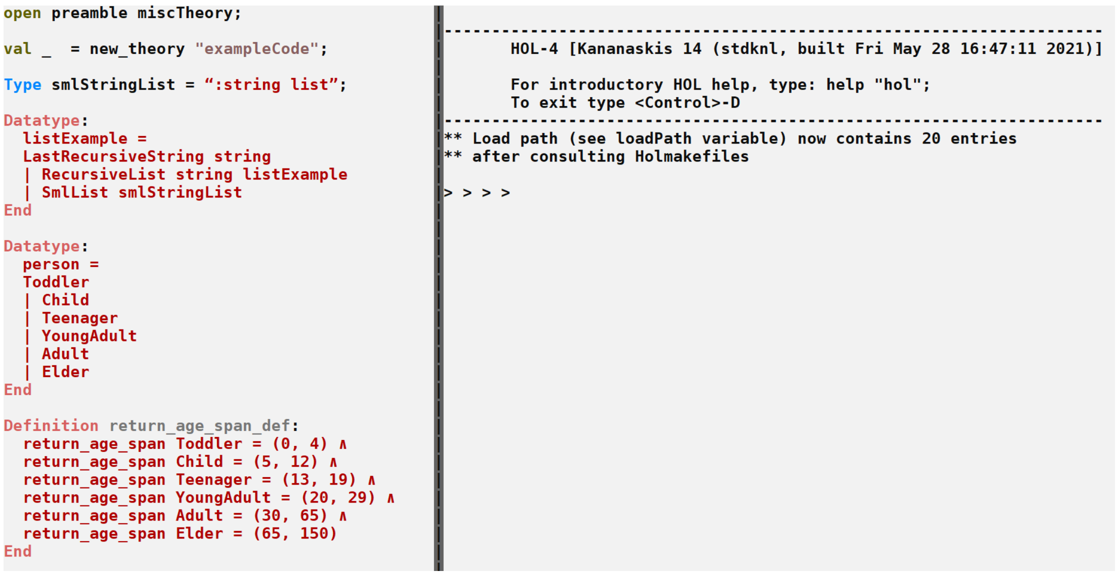
The CakeML ecosystem includes a verified checker, *cake_lpr*, for checking the output of a SAT solver [6]. It checks proof traces, which are generated by a SAT solver when the SAT problem is unsatisfiable. If a verified checker finds a fault in the proof-trace, it means that a bug exists in the SAT solver [7, 8].

If the input to the SAT solver is satisfiable, no checker is needed because the verification is trivial [7, 8]. Instead of a proof trace, the SAT solver outputs an assignment of the used variables. It is enough to test the proposed assignments and see if the CNF-formula evaluates to true.

2.3 Formal Verification and HOL4

HOL4 is a proof assistant specialized for higher-order logic, which uses Standard ML (SML) as the implementation language [9]. It provides built-in libraries and proof tools to help implement new proof tools and solve theorems automatically or interactively [10].

When using HOL4, there are two windows, which can be seen in Figure 2.1. In the left window, a proof script is developed. The script consists of commands such as `Type`, `Datatype`, `Definition`, `Theorem`, and tactics. Tactics are the methods used to prove theorems. Most of the code is similar to other functional programming languages. The right window shows how HOL4 responds to these commands. The code is loaded from the left window into the right.



```
open preamble miscTheory;

val _  = new_theory "exampleCode";

Type smlStringList = ":string list";

Datatype:
  listExample =
  LastRecursiveString string
  | RecursiveList string listExample
  | SmlList smlStringList
End

Datatype:
  person =
  Toddler
  | Child
  | Teenager
  | YoungAdult
  | Adult
  | Elder
End

Definition return_age_span_def:
  return_age_span Toddler = (0, 4) ∧
  return_age_span Child = (5, 12) ∧
  return_age_span Teenager = (13, 19) ∧
  return_age_span YoungAdult = (20, 29) ∧
  return_age_span Adult = (30, 65) ∧
  return_age_span Elder = (65, 150)
End
```

```
---------------------------------------------------------------------
       HOL-4 [Kananaskis 14 (stdknl, built Fri May 28 16:47:11 2021)]

       For introductory HOL help, type: help "hol";
       To exit type <Control>-D
---------------------------------------------------------------------
** Load path (see loadPath variable) now contains 20 entries
** after consulting Holmakefiles

> > > >
```

Figure 2.1: An example of the workspace for HOL4.

One thing that differs from other functional programming languages is theorems and proofs. Theorems are written using logical operators. HOL4 proofs are a collection of tactics that will be applied to the theorem to prove its validity.

The code below shows a theorem and proof, saying that a toddler has an age between 0 and 4.

```
Theorem toddler_age_is_correct:
 ∀ person.
 person = Toddler ⟹
 (return_age_span person = (0, 4))
Proof
 Cases_on‘person’
 >> rw[return_age_span_def]
QED
```

Figure 2.2 shows the same theorem and proof loaded into HOL4.

```
> # # # # # val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∀person. person = Toddler ⇒ return_age_span person = (0,4)
   : proofs
> val it = (): unit
> > > > > # # # OK..
val it =
   Initial goal proved.
   ⊢ ∀person. person = Toddler ⇒ return_age_span person = (0,4): proof
```

Figure 2.2: The theorem toddler_age_is_correct loaded into HOL and proved.

When a theorem is loaded into HOL4, it is called a goal. A goal is proved using tactics, such as `Cases_on` and `rw[]` above. `rw`, which is short for rewrite, is used to simplify proofs. Inside the brackets, one specifies already proved theorems or definitions that will be used to prove the goal. In the example above, the definition of the function `return_age_span` is used. >> is used as a link between two tactics.

3

Sudoku Example

This chapter describes how our datatypes can be used to solve the example sudoku from Chapter 1. It also describes how we can determine if the sudoku is valid. A sudoku is valid if it has *exactly one* solution. If the sudoku is valid, the problem of finding a *second* solution will be unsatisfiable.

3.1 The Sudoku Type in HOL4

First, we need to implement a type for sudoku in HOL4. We implement a cell as a number, with empty cells represented by the value 0. A row is implemented as a list of cells, and a sudoku is implemented as a list of rows. In HOL4, the type of a sudoku is then `num list list`. Here is the unsolved version of our example from Figure 1.1 implemented in HOL4.

```
[[5; 3; 0; 0; 7; 0; 0; 0; 0];
 [6; 0; 0; 1; 9; 5; 0; 0; 0];
 [0; 9; 8; 0; 0; 0; 0; 6; 0];
 [8; 0; 0; 0; 6; 0; 0; 0; 3];
 [4; 0; 0; 8; 0; 3; 0; 0; 1];
 [7; 0; 0; 0; 2; 0; 0; 0; 6];
 [0; 6; 0; 0; 0; 0; 2; 8; 0];
 [0; 0; 0; 4; 1; 9; 0; 0; 5];
 [0; 0; 0; 0; 8; 0; 0; 7; 9]]
```

We also need to define what it means for a sudoku to be well-formed. This is done in the function sudoku_ok,

$$\begin{aligned}&\mathsf{sudoku\_ok}\ \mathit{sudoku} \overset{\mathsf{def}}{=}\\ &\quad \mathsf{length}\ \mathit{sudoku} = 9 \wedge \mathsf{every}\ (\lambda\, \mathit{row}.\ \mathsf{length}\ \mathit{row} = 9)\ \mathit{sudoku} \wedge\\ &\quad \mathsf{every}\ (\lambda\, n.\ 0 \leq n \wedge n \leq 9)\ (\mathsf{flat}\ \mathit{sudoku}).\end{aligned}$$

This function consists of three parts. The first part, length $\mathit{sudoku} = 9$, tells that the length of the sudoku should be nine. Remember that a sudoku is a list of rows, so this constraint says that a sudoku must have nine rows.

The second part, every $(\lambda\, \mathit{row}.$ length $\mathit{row} = 9)$ sudoku says that the length of each row should be nine. every takes a predicate and checks that it holds for all elements in a list. Here, the list is the sudoku where every element is a row. The predicate, $\lambda\, \mathit{row}.$ length $\mathit{row} = 9$, checks that the length of the row is equal to nine.

The last part, every $(\lambda\, n.\ 0 \leq n \wedge n \leq 9)$ (flat sudoku), checks that all cells have a valid value. flat takes a list of lists and flattens it to simply one list. Thus, flat sudoku returns one list with all cells. The predicate checks that an element has a value between 0 and 9.

3.2 Encoding

The idea of the encoding is to have one versatile datatype, to which the sudoku can be easily translated. This datatype should be able to capture a variety of logical problems. From this datatype, we encode the problem to CNF via several intermediate datatypes. This process is visualized in Figure 3.1.

Passing the encoded problem to a SAT solver gives a solution in the form of an assignment function. After encoding this assignment function back to match the sudoku datatype, we will use it to solve the sudoku.

It is important that all these encodings, both of the problem and the solution, are proved to preserve satisfiability. By this, we mean that if the sudoku has a solution, the SAT problem should be satisfiable. Moreover, if the sudoku does not have a solution, the SAT problem should be unsatisfiable.

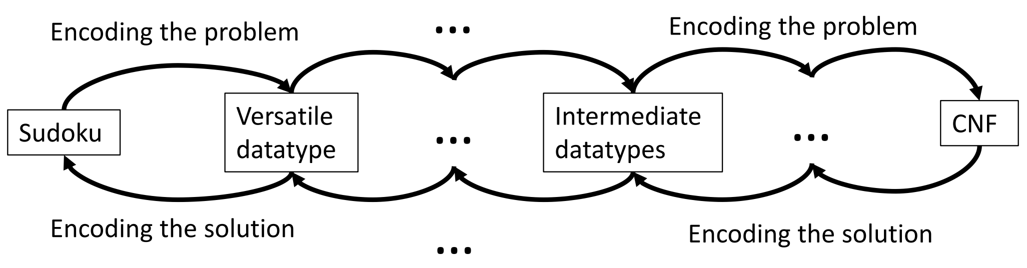


Figure 3.1: The steps of the encoding process, first encoding the problem from sudoku to CNF and then encoding the solution back to sudoku.

3.3 Satisfiability

The assignment function used to solve the sudoku will have the type `num` $\rightarrow$ `num`, which means it is a function from number to number. The input is the position of a cell, and the output is the value that should be assigned to that cell. The positions are numbered from 00 to 88, where the first number represents the row and the second number represents the column. An assignment is valid if it assigns every position to a value between 1 and 9.

Deciding whether an assignment function solves a sudoku consists of three steps:

1. Generating lists of all rows, columns, and blocks.
2. Filling each cell in the lists of rows, columns, and blocks, using the assignment function.
3. Checking if all values are distinct in each of the lists.

If all values are distinct within each of these lists, the assignment function provides a solution to the sudoku.

3.4 Proving That a Sudoku is Valid

As previously mentioned, a sudoku is valid if it has exactly one solution. Proving that a sudoku is valid is done in two steps. First, we solve the sudoku. Then we try to solve the sudoku again but specify that we want a different solution. This specification is done by adding the constraints that at least one cell should have a different value than in the first solution. Using our example sudoku and solution in Figure 1.1, we would specify that cell 02 should not have value 4, or cell 03 should not have value 6, and so on. The reason that we use disjunction between the different constraints is that we only want to forbid this *exact* solution, not all solutions where cell 02 has value 4.

If we find a solution the first time but not a different solution the second time, then the sudoku is valid. For this to hold, we must prove that the encoding preserves satisfiability. In other words, we need to prove that the sudoku has a solution if and only if the SAT problem is satisfiable. Logically this can be expressed as

$$\begin{aligned}&\mathsf{eval\_sudoku}\ w\ sudoku \iff \\ &\mathsf{eval\_sat}\ (\mathsf{encode\_assignment}\ w)\ (\mathsf{encode\_sudoku}\ sudoku),\end{aligned} \tag{3.1}$$

where w is an assignment function.

In the following chapter, we will look at how the intermediate datatypes can be implemented in order to build up a versatile datatype to encode a variety of logical problems. Our result will be three different versatile datypes, suitable for different types of logical problems. Throughout all intermediate datatypes, we need to prove a version of equation (3.1). Proving preservation of satisfiability is the key to freely use our datatypes to encode SAT problems.

4

Verified Encoding Functions

This chapter explains the development of the verified encoding functions. Each encoding function corresponds to a new datatype with added functionality. Our approach is to add functionality in small steps rather than starting with one datatype including everything.

4.1 The Encoding Functions at a Glance

Figure 4.1 shows the implemented datatypes and how they relate to each other. The datatypes in bold are `cnf` and the three main datatypes, which should be used to encode problems. `cnf` is in bold because it is the datatype to which all other datatypes will be encoded.

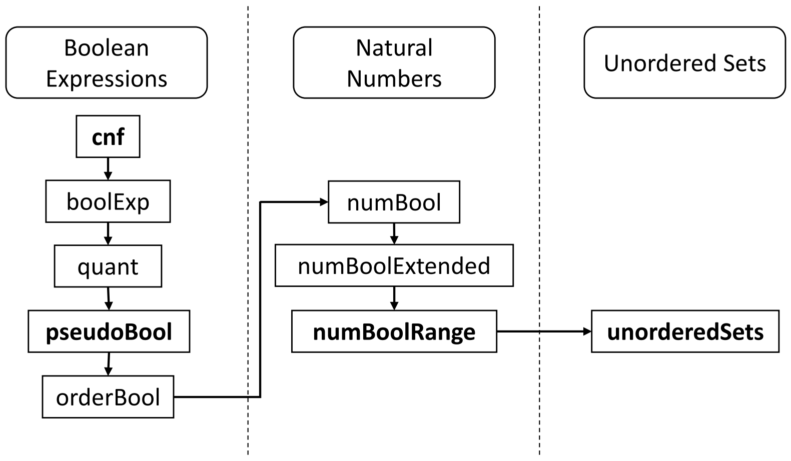


Figure 4.1: Implemented datatypes and their relations.

The left-hand side of the figure shows all datatypes for Boolean expressions. `boolExp` extends `cnf` with implication and equivalence and allows all operators to be used in any order. The next datatype adds the existential (∃) and universal (∀) quantifier, which will quantify over Booleans. The datatype `pseudoBool` has added operators for at least one, at most one, and exactly one of a list of Boolean expressions. `pseudoBool` is the main datatype for Boolean expressions.

After this follows the encoding of natural numbers. First is the datatype `orderBool`. It contains only Boolean expressions but prepares for natural numbers. After this follow three datatypes for natural numbers that each adds additional functionality.

To the right in the figure is the encoding of unordered sets. It only has one datatype, which builds on natural numbers.

4.2 The CNF Datatype

The CNF datatype will be used as the base, to which all other datatypes will be encoded. Its smallest building block is the literal. A literal is either a Boolean variable or the negation of a Boolean variable.

We use literals to build clauses. We define the clause datatype as

```
clause =
   ClauseEmpty
  | ClauseLit literal
  | ClauseOr clause clause.
```

A clause can either be empty, a literal or a disjunction of clauses.

A CNF consists of conjunctions of clauses. We define the datatype as

```
cnf = CnfEmpty | CnfClause clause | CnfAnd cnf cnf.
```

A CNF can either be empty, a clause, or a conjunction of CNFs.

The evaluation function for CNF is called `eval_cnf`. It takes a CNF formula and an assignment function w as arguments, where w is a function from Boolean variables to Booleans. CnfAnd and ClauseOr are evaluated as the logical operators $\wedge$ and $\vee$, and a literal is evaluated using the assignment function. If bv is the Boolean variable of the literal, the evaluation of bv is $w\ bv$.

Since we want

$$\texttt{eval\_cnf}\ (\mathsf{CnfAnd}\ c\ \mathsf{CnfEmpty}) = \texttt{eval\_cnf}\ c,$$

we let CnfEmpty evaluate to true. For similar reasons with ClauseOr, we let ClauseEmpty evaluate to false.

4.3 Encoding Booleans

We encode Boolean expressions using several datatypes, adding more functionality in every step. First, we encode simple Boolean expressions containing True, False, literals, and the operators not, and, or, implication, and equivalence. This is done in two different ways; first a naive encoding and then a smarter one resulting in the CNF formula being linear in size relative to the input size. After this, we add the existential and universal quantifiers in a new datatype. Then we add pseudo-Boolean expressions. Lastly, we add an operator specialized for the encoding that we will use for natural numbers.

Since type constructors need to be unique in HOL4, we have added a prefix to most of the constructors in the datatypes in the following sections. For example, the And constructor in the quantifier datatype is called QAnd.

4.3.1 Naive Encoding of Simple Boolean Expressions

Our first goal is to encode Boolean expressions into CNF. In HOL4 the datatype is defined as

```
boolExp =
   True
  | False
  | Lit literal
  | Not boolExp
  | And boolExp boolExp
  | Or boolExp boolExp
  | Impl boolExp boolExp
  | Iff boolExp boolExp.
```

To encode Boolean expressions to CNF we follow a simple algorithm described in the book Logic in Computer Science [4]. The first step is to remove all implications and equivalences using

$$\begin{aligned} b_1 &\implies b_2 = \neg b_1 \lor b_2 \\ b_1 &\iff b_2 = (b_1 \land b_2) \lor (\neg b_1 \land \neg b_2). \end{aligned}$$

The second step is to push all negations inwards so that the only negated expressions are literals. After this is done, the expression is said to be in negation normal form (NNF). This is achieved using de Morgan's laws,

$$\begin{aligned} \neg(b_1 \land b_2) &= \neg b_1 \lor \neg b_2 \\ \neg(b_1 \lor b_2) &= \neg b_1 \land \neg b_2. \end{aligned}$$

The last part is to encode from NNF to CNF. True is encoded to CnfEmpty and False is encoded to ClauseEmpty. Lit is already in CNF. $b_1 \land b_2$ is also in CNF if both b_1 and b_2 are in CNF.

To encode $b_1 \lor b_2$, we first ensure that both b_1 and b_2 are in CNF. Then there are two cases. The first case is when both b_1 and b_2 are Booleans or literals, which results in $b_1 \lor b_2$ already being in CNF. The second case is when at least one of them is a conjunction. This case is encoded using the distribution laws. If for example $b_1 = (b_{11} \land b_{12})$, we use

$$(b_{11} \land b_{12}) \lor b_2 = (b_{11} \lor b_2) \land (b_{12} \lor b_2),$$

to get an expression that is in CNF.

After defining our encoding function, we need to prove that it preserves satisfiability. For this, we need the function eval_boolExp. This function is constructed by replacing the datatype constructor with the corresponding logical operator. For example, `And` is replaced by $\land$. This function is then used in the theorem

$$\vdash \mathsf{eval\_boolExp}\ w\ b \iff \mathsf{eval\_cnf}\ w\ (\mathsf{boolExp\_to\_cnf}\ b)$$

that proves satisfiability.

4.3.2 Improving the Encoding of Simple Boolean Expressions

Our previously described encoding of Boolean expressions has one problem; the encoding of some types of expressions leads to an exponential increase in size [11].

Therefore, we implement an encoding function from Boolean expressions to CNF based on Tseytin transformation.

The idea of Tseytin transformation is to replace each subformula with a fresh Boolean variable. Then we add the encoding that the variable and the subformula are equal [11].

Say that we want to encode

$$(b_1 \land b_2) \lor (b_3 \land b_4),$$

where $b_1, ..., b_4$ are Boolean variables. The first subformula is $b_1 \land b_2$, which we replace with b_5. We replace the second subformula $b_3 \land b_4$ with b_6. Then the whole formula can be written as $b_5 \lor b_6$, which we replace by b_7.

The encoding will then be built up of the parts

$$\begin{aligned}
&b_5 \iff (b_1 \land b_2)\\
&b_6 \iff (b_3 \land b_4)\\
&b_7 \iff (b_5 \lor b_6)\\
&b_7,
\end{aligned}$$

where the last row means that the variable b_7 should be true.

Since we cannot use equivalence in a CNF formula, we also need to encode all of the substitutions. For this, we define one encoding function for each of the possible subformulas. For example, we can rewrite the first substitution above using

$$\begin{aligned}
&b_5 \iff (b_1 \land b_2) =\\
&(b_5 \implies (b_1 \land b_2)) \land ((b_1 \land b_2) \implies b_5) =\\
&(\neg b_5 \lor (b_1 \land b_2)) \land (\neg(b_1 \land b_2) \lor b_5) =\\
&(\neg b_5 \lor b_1) \land (\neg b_5 \lor b_2) \land (\neg b_1 \lor \neg b_2 \lor b_5).
\end{aligned}$$

The last row above is used as the CNF version of the encoding of substitution of a conjunction. Similar encoding functions are defined for all substitutions, and then combined to encode Boolean expressions using Tseytin transformation.

4.3.3 Quantifiers over Boolean Expressions

The next step is to implement Boolean quantifiers. Two different quantifiers are implemented: the universal quantifier ($\forall$) and the existential quantifier ($\exists$). To quantify over Booleans is to quantify over the set $\{\mathrm{T}, \mathrm{F}\}$. Therefore, $\forall x.\ p(x)$ means that $p(x)$ should be true *both* when x is replaced with true and replaced with false. Similarly, $\exists x.\ p(x)$ means that $p(x)$ should be true *either* when x is replaced with true or replaced with false.

The implementation of quantifiers can be found in a datatype called `quant`,

```
quant =
  ...
  | QAll name quant
  | QEx name quant,
```

where `name` is the variable the quantifier is applied to.

We evaluate quantifiers by adding mappings of the Boolean variable in the assignment function w. If we want to evaluate QAll b_1 $((b_1 \vee b_2) \wedge (b_1 \wedge b_3))$, we need to evaluate a conjunction between the two cases. The evaluation becomes

$$\begin{gathered}\text{eval } w \text{ QAll } b_1\ ((b_1 \vee b_2) \wedge (b_1 \wedge b_3)) \iff \\ \text{eval } w(\!|b_1 \mapsto T|\!)\ ((b_1 \vee b_2) \wedge (b_1 \wedge b_3)) \ \wedge\ \text{eval } w(\!|b_1 \mapsto F|\!)\ ((b_1 \vee b_2) \wedge (b_1 \wedge b_3)).\end{gathered}$$

The `quant` datatype is encoded to `boolExp` in a way that is similar to the evaluation process. The difference is that instead of mapping the variable in w, we replace the variable in the expression.

4.3.4 Pseudo-Boolean Constraints

Pseudo-Boolean constraints are constraints that consists of addition of Boolean variables. Three cases of pseudo-Boolean constraints commonly used in puzzles are the *at least one constraint*, *at most one constraint* and *exactly one constraint.* They look like

$$\begin{aligned}\text{At least one: } & x_1 + ... + x_n \geqslant 1 \\ \text{At most one: } & x_1 + ... + x_n \leqslant 1 \\ \text{Exactly one: } & x_1 + ... + x_n = 1.\end{aligned}$$

We include these three pseudo-Boolean constraints in our formal syntax as the following cases of a datatype for Boolean expressions extended with pseudo-Boolean constraints,

```
pseudoBool =
  ...
  | PLeastOne (pseudoBool list)
  | PMostOne (pseudoBool list)
  | PExactlyOne (pseudoBool list).
```

The implementations consist of a list of Boolean expressions. The semantics are defined to be as in the equations above. The procedure of evaluating an expression consists of the following steps:

1. Evaluate each Boolean expression to T and F
2. Map T and F to 1 and 0
3. Add the values and compare them to 1:
   - PLeastOne is true for $\geqslant 1$
   - PMostOne is true for $\leqslant 1$
   - PExactlyOne is true for $= 1$

Next, we have to define the encoding of our pseudo-Boolean constraints. This is done differently for the three instances.

Say that we want to encode PLeastOne $[b_1;\ b_2;\ b_3]$. At least one of them has to be true, but we can also have two or more to be true. This is the same as having b_1 or b_2 or b_3. I.e., the encoding is $b_1 \vee b_2 \vee b_3$.

If we instead want to encode PMostOne $[b_1;\ b_2;\ b_3]$, we get one clause for each expression. If b_1 is true, both b_2 and b_3 are false. This can be expressed as $b_1 \implies (\neg b_2 \wedge \neg b_3)$. Note that it also holds if b_1 is false. To capture the case when b_1 is false and b_2 is true, we add the clause $b_2 \implies \neg b_3$. Then we have the case when both b_1 and b_2 are false. In this case, it does not matter whether b_3 is true or false, which we write as $b_3 \implies T$. The final encoding becomes $(b_1 \implies (\neg b_2 \wedge \neg b_3)) \wedge (b_2 \implies \neg b_3) \wedge (b_3 \implies T)$.

The encoding of exactly one, for example PExactlyOne $[b_1;\ b_2;\ b_3]$, is the simplest one yet. It is defined as at least one *and* at most one. For our example that is PLeastOne $[b_1;\ b_2;\ b_3] \wedge$ PMostOne $[b_1;\ b_2;\ b_3]$.

4.3.5 Preparing the Boolean Datatype for Encoding of Natural Numbers

In our encoding of natural numbers, we will use *order encoding* [12]. This is an encoding method where the variable $x \in [0, k]$ is encoded by b_0, ..., b_k, where b_i is true if $x \leqslant i$. For example, if $x \in [0, 2]$ then $x = 1$ is represented by [F, T, T].

From the order of natural numbers, it follows that if $x \leqslant i$ then $x \leqslant i+1$. However, if $x \nleqslant i$ then we do not know whether $x \leqslant i + 1$ or not. Therefore, $b_i \implies b_{i+1}$.

To make the encoding of natural numbers easier, we create a datatype that captures this ordering. This datatype is called `orderBool`. It is an extension of `pseudoBool`, with the added constructor `OrderAxiom` that takes a list of Boolean variables as an argument.

`OrderAxiom` evaluates to true if, after evaluating each Boolean variable individually, the list consists of 0 or more false followed by 1 or more true. For example, the list [F, T, T] evaluates to true, but the list [F, T, F] evaluates to false. The last example says that $x \leqslant 1$ and $x \nleqslant 2$, which cannot be true at the same time.

The reason that we need at least one value that evaluates to true in the list is to keep the number variable inside the allowed range. For example, if $x \in [0, 2]$ is represented by [F, F, F], the representation says that $x > 2$ which is not allowed.

The encoding of `OrderAxiom` is defined as

$$\left(\bigwedge_{i=0}^{k-1} (b_i \implies b_{i+1})\right) \wedge b_k.$$

4.4 Encoding Natural Numbers

The next step is to encode natural numbers. As with the Boolean expressions, we do this sequentially through several datatypes. The first datatype is called `numBool` and adds the possibility of having natural numbers together with four core operators. Each natural number belongs to the same range of valid numbers, from 0 up to some specified number k. The next datatype is called `numBoolExtended`, and it extends `numBool` with more operators. The last datatype is called `numBoolRange`, and it allows variables to belong to different ranges.

This section describes the datatypes, evaluation, and encoding functions.

4.4.1 The First Datatype for Natural Numbers

The datatype `numBool` extends the usability of the datatype of Boolean expressions. This means that it allows usage of true, false, Boolean variables, and the operators not, and, or, implication, and equivalence. It also contains the operators

$$\begin{aligned}\text{Add: } & x + y = z\\ \text{Leq: } & x \leqslant y\\ \text{EqConst: } & x = n\\ \text{LeqConst: } & x \leqslant n,\end{aligned}$$

where x, y and z are variables of the type natural number, and n is a constant.

Note that the four added operators all evaluate to either true or false, which means that they can be used freely in any expressions. We could, for example, write the expression

$$b \Longrightarrow (x \leqslant 10),$$

and then the program will try to find a Boolean value for b and a natural number for x that satisfies the equation.

The evaluation function for `numBool` uses two different assignment functions; one for Boolean variables and one for number variables. The assignment function for Boolean variables is defined in the same way as in previous evaluations, and the assignment function for natural numbers is new. If we let w' be the new assignment function, the evaluation of the new operators becomes

$$\begin{aligned}\text{Add: } & w'\ x + w'\ y = w'\ z\\ \text{Leq: } & w'\ x \leqslant w'\ y\\ \text{EqConst: } & w'\ x = n\\ \text{LeqConst: } & w'\ x \leqslant n.\end{aligned}$$

4.4.2 Encoding Using Order Encoding

We encode `numBool` to the datatype `orderBool`, using *order encoding.* This means that the variable $x \in [0, k]$ is encoded by b_0, ..., b_k where b_i is true if $x \leqslant i$ [12].

The encoding consists of four steps:

1. Create a *variable map* that maps every number variable x to the corresponding Boolean variables b_i used for the encoding of x.
2. Encode the expression.
3. Encode the axioms, which specifies that natural numbers have an order. We do this with a direct application of the constructor `OrderAxiom`, described in Section 4.3.5.
4. Encode the new assignment function w' into assignments from Boolean variables to Booleans.

To create the variable map, we recurse through the expression and create a list of all the used number variables. The variable map is then constructed as a list of pairs, where each pair contains a number variable and a list of fresh Boolean variables used for the encoding.

In order to encode the expression, we need to define new encoding functions for our four added operators. The easiest one to encode is LeqConst, since it corresponds well to the definition of order encoding. The equation $x \leqslant i$ is encoded as b_i.

The encoding of EqConst is almost as simple. $x = i$ can be rewritten to $x \leqslant i \wedge \neg(x \leqslant i - 1)$, which is then encoded as $b_i \wedge \neg b_{i-1}$.

The encoding of $x \leqslant y$, where both x and y are variables, builds on the concept that if $y \leqslant i$, then $x \leqslant i$. If we let b_0^x, ..., b_n^x be the Boolean variables corresponding to x and b_0^y, ..., b_n^y be the Boolean variables corresponding to y, the encoding becomes

$$\bigwedge_{i=0}^{k} (b_i^y \implies b_i^x) .$$

The encoding of $x + y = z$ builds on two concepts. The first concept is that $x+y = z$ holds if and only if for all i, $(x+y \leqslant i) \iff (z \leqslant i)$. This can be written as

$$\bigwedge_{i=0}^{k} ((x + y \leqslant i) \iff (z \leqslant i)) . \tag{4.1}$$

We already know how to encode $z \leqslant i$, so it remains to encode $x + y \leqslant i$.

The encoding of $x + y \leqslant i$ is the second concept. $x + y \leqslant i$ will be true if we have *one combination* of x and y that is smaller than i. For example, we could have $x \leqslant 0$ and $y \leqslant i$, or $x \leqslant 1$ and $y \leqslant i - 1$. The encoding function becomes

$$\bigvee_{j=0}^{i} \left(b^x_j \wedge b^y_{i-j}\right) . \tag{4.2}$$

Combining equation (4.1) with equation (4.2) we get the full equation for encoding $x + y = z$,

$$\bigwedge_{i=0}^{k} \left(\left(\bigvee_{j=0}^{i} \left(b^x_j \wedge b^y_{i-j}\right) \right) \iff b^z_i \right)$$

We encode the assignment function for number variables, w', by modifying the encoding function for Boolean variables. Assume that w is the encoding function for all Boolean variables present in our expression. Then the encoded assignment function will do the following when applied to a Boolean variable b. If b is present in the variable map, representing $x \leqslant v$ for some x and v, return $w'\ x \leqslant v$. Else, return $w\ b$.

4.4.3 Extending the Datatype for Natural Numbers

The next step is the datatype `numBoolExtended`, where constructors are added for all expressions of the forms

$$x \# y$$
$$x \# n$$
$$n \# x,$$

where x and y are number variables, n is a constant and # is one of $=$, $\neq$, $<$, $\leqslant$, $>$ and $\geqslant$. For example, a constructor for $x < y$ is added.

The evaluation of the added constructors is made by applying w' to all number variables. In the example above, the evaluation becomes $w'\ x < w'\ y$.

In the encoding function, all of the added constructors are encoded into combinations of Leq, EqConst, and LeqConst from `numBool`. For example, $x < y$ is encoded to $\neg(y \leqslant x)$.

4.4.4 Adding Different Ranges for Different Variables

In `numBool` and `numBoolExtended`, all number variables ranged from 0 to some k, where k was provided together with the equation. The next step is to allow different variables to have different ranges. The input provided together with the equation is then a range list, which is a list of all number variables and their upper and lower bounds. However, only ranges of non-negative numbers are allowed.

In order for the evaluation to work, w' must evaluate all number variables to a value within the allowed range. Aside from that extra condition, the evaluation is done in the same way as before.

For the encoding part, the strategy is to encode the ranges as part of the equation. A range $x \in [m, n]$ is replaced by $(m \leqslant x) \wedge (x \leqslant n)$. We let k be the highest upper bound among all variables.

4.5 Encoding Unordered Sets

The third versatile datatype is called `unorderedSets`. In this datatype, both variables and the sets they belong to are specified. The datatype is called unordered sets because the logic does not contain any ordering relation between the elements of the set.

This datatype can have both Boolean variables and set variables. The datatype is an extension of the `boolExp`, with two added operators. `EqVarCon` is equality between a set variable and a constant. `EqVarVar` is equality between two set variables. Every time a set variable is used in an equation, one needs to specify the name of the set that the variable belongs to. The equation is not valid if it contains equalities between variables that belong to different sets.

Together with an equation to be solved, one also needs to specify the sets and their elements. This is done in a list, where each element is a pair of the set name and a list of the set elements.

We do the evaluation using two different assignment functions. The first one is a function from Boolean variables to Booleans, which we have used before. The second one is a function from set variables to set elements. The new equality constructors are evaluated by evaluating each set variable and checking the equality.

We encode this datatype to `numBoolRange`. This means that each element in a set is encoded to a natural number. Since elements in a set have a fixed order in the set list, we use the index of each element for the encoding.

Three things need to be encoded; the equation, the set list, and the assignment function for set variables. In the equation, the only new constructors are `EqVarCon` and `EqVarVar`. `EqVarCon` is encoded to equality between the variable and the index of the constant in the set list. `EqVarVar` is encoded to equality between two number variables.

The set list is encoded to a range list. This is done by letting each variable range from 0 to the highest index in the set list.

The assignment function for set variables is encoded to an assignment function for number variables. If w' is the assignment function for set variables and x is a set variable, $w'\ x$ is the element of x. The new assignment function is encoded to return the *index* of $w'\ x$ in the set list.

5

Proving That the Encodings Preserve Satisfiability

In the previous chapter, we defined the datatypes of our encoding chain. For each datatype, we also defined one function for evaluation and one function for encoding. In this chapter, we use the encoding and the evaluation functions to prove that the encodings preserve satisfiability.

5.1 The Satisfiability Theorem

By proving that the encodings preserve satisfiability, we mean two things. Firstly, if the SAT solver gives a solution, we also have a solution to the original problem. Secondly, if the SAT solver concludes that the problem is unsatisfiable, the original problem is unsatisfiable.

We achieve both of these things by proving that the original problem and the encoded problem evaluates to the same value, for every assignment function w. For example, for the encoding from `pseudoBool` to `cnf` we want to prove

$$\vdash \mathsf{eval\_pseudoBool}\ w\ b \iff \mathsf{eval\_cnf}\ w\ (\mathsf{pseudoBool\_to\_cnf}\ b),$$

where b is any expression of type `pseudoBool` and w is any assignment.

In order to simplify the proofs, we do one proof for each encoding step. For example, instead of proving the satisfiability theorem directly from `pseudoBool` to `cnf`, we first prove the theorem from `pseudoBool` to `quant`,

$$\vdash \mathsf{eval\_pseudoBool}\ w\ b \iff \\ \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ b).$$

After that, we prove the satisfiability theorem for the encoding from `quant` to `boolExp`, and then from `boolExp` to `cnf`. These proofs are then combined to create the original proof, that the encoding of the problem to CNF preserves satisfiability.

5.2 Proof by Induction

The key idea in all proofs is to use induction on the expression. In the examples above, the expression is b, which is of type `pseudoBool`. The induction will generate a set of base cases, and a set of inductive cases.

The base cases will come from the non-recursive constructors, such as the constructors for true, false and literals. The base cases are often simple to solve, using a rewrite tactic with the appropriate definitions. For example, the case

$$\begin{aligned}&\vdash \mathsf{eval\_pseudoBool}\ w\ \mathtt{PTrue} \iff \\ &\quad \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ \mathtt{PTrue}),\end{aligned}$$

where `PTrue` is the true-constructor for `pseudoBool`, is solved by
`rw[pseudoBool_to_quant_def, eval_pseudoBool_def, eval_quant_def]`.

This `rw` contains three definitions. The encoding from `pseudoBool` to `quant` will rewrite $\mathsf{pseudoBool\_to\_quant}\ \mathtt{PTrue}$ to `QTrue` that is the true-constructor for `quant`. The evaluation functions for both datatypes, `eval_pseudoBool_def` and `eval_quant_def`, evaluates both sides of the theorem. In this case, both sides evaluates to true. Since both sides evaluates to the same value, this case is proved.

The inductive steps are proved in a similar way using rewrites. The difference here is that we have inductive hypotheses. For example, in the case

$$\begin{aligned}&\vdash \mathsf{eval\_pseudoBool}\ w\ (\mathtt{PAnd}\ b_1\ b_2) \iff \\ &\quad \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ (\mathtt{PAnd}\ b_1\ b_2)),\end{aligned}$$

where b_1 and b_2 are arbitrary expressions of type `pseudoBool`, we have the assumptions that the theorem holds for b_1 and b_2,

$$\begin{aligned}&\vdash \mathsf{eval\_pseudoBool}\ w\ b_1 \iff \\ &\quad \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ b_1)\end{aligned}$$

and

$$\begin{aligned}&\vdash \mathsf{eval\_pseudoBool}\ w\ b_2 \iff \\ &\quad \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ b_2).\end{aligned}$$

If we use the same `rw` as above, the goal changes to

$$\begin{aligned}\vdash\ & \mathsf{eval\_pseudoBool}\ w\ b_1 \land \mathsf{eval\_pseudoBool}\ w\ b_2 \iff \\ & \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ b_1) \land \\ & \mathsf{eval\_quant}\ w\ (\mathsf{pseudoBool\_to\_quant}\ b_2),\end{aligned}$$

which holds from the assumptions.

5.3 Adding Fresh Variables

In some of the encoding functions, we add fresh variables. For example, this is the case when we encode from `boolExp` to `cnf` using Tseytin transformation. Adding fresh variables complicates the proof, because we cannot use the same assignment function on both sides. For example, an assignment function that assigns all variables in the expression of type `boolExp` correctly might not assign the fresh variables in the expression of type `cnf` correctly.

To solve this problem, we decided to encode the assignment function. This means that we get two encoding functions, one for the expression and one for the assignment function. When encoding the assignment function, we let each fresh variable be assigned to the value of the subformula that it replaces.

5.4 Proving the Encoding of Natural Numbers

Another encoding where we add fresh variables is the encoding from `numBool` to `orderBool`. Here the encoding of the assignment function must go from assignments of number variables to assignments of Boolean variables. We also need to be able to encode the solution from the SAT solver, so therefore we need a second encoding of the assignment function, back to numbers. Thus we define two encodings of assignments, one from Booleans to numbers and one from numbers to Booleans. To be certain that these functions behave correctly, we also prove that if you apply both of the encodings to an assignment function, you end up with the same assignment function as you started with.

A strategy that we use in the satisfiability proofs from `numBool` to `orderBool` is to separate the creation and the usage of the variable map. The variable map contains number variables and the fresh Boolean variables corresponding to the number variables. We know that after creating a variable map from the expression b, the variable map contains exactly those number variables that are present in b. However, the encoding of the expression would work just fine even if there were additional, unused number variables in the variable map. Moreover, the inductive steps in the encodings become more manageable if we can use the same variable map everywhere.

In the proof, we start with proving that the created variable map is *well-formed*. For example, it should contain all variables in the current expression and contain no duplicate variables. Then, for the rest of the proof, we assume that we have a well-formed variable map.

Another strategy in the satisfiability proofs for natural numbers is to separate the proof into one lemma about the axioms and one lemma about the expression. This is the same way as the encoding was divided. First, we prove that the axioms are *always* true. Then, we prove that the encoding of the expression preserves satisfiability, assuming that the axioms are true.

6

Case Studies

This chapter shows some problems and how they are encoded into the three versatile datatypes. For most problems, the encoding to one of the versatile datatypes is trivial, since the datatypes include the necessary functionality to easily capture the problem. The problems are divided into two different categories; algorithmic problems and logical puzzles.

6.1 Algorithmic Problems

This section explores different famous algorithmic problems and how these can be encoded to one of the three datatypes.

6.1.1 N-Queens Problem

The n-queens problem is a problem commonly used in benchmarks for hardware processors [13], and to compare the performance of different SAT-solvers [14]. The problem is to place n queens on a chessboard of the size $n \times n$ so that no two queens threaten each other. Thus, a solution requires that two queens do not share the same row, column, or diagonal. Figure 6.1 shows one out of 92 solutions to the eight-queens problem.

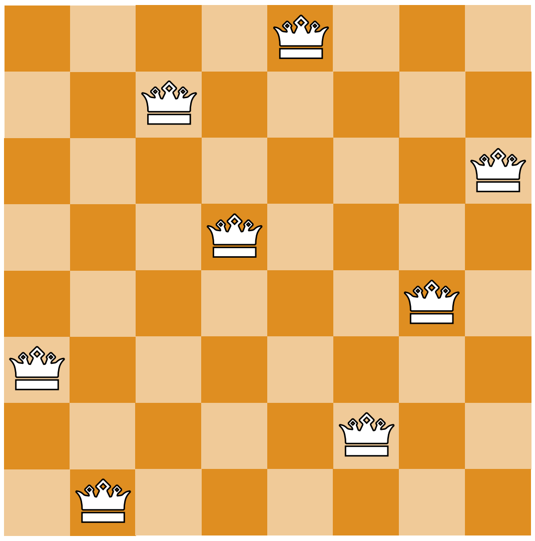


Figure 6.1: A solution to the eight-queens problem.

We encoded this problem to the datatype `pseudoBool`. Since the *at most one* and the *at least one* constraints are defined in `pseudoBool`, this encoding is relatively straightforward. Each square is represented by a Boolean variable that is true if there is a queen in that square and false otherwise. Each row, column, and diagonal is encoded to an at most one constraint since the queens should not be able to attack each other. Additional to the at most one constraints, the problem is only solved when n queens are placed on the board. To guarantee this, a constraint to have at least one queen per row is added. All constraints are combined with conjunction since all of them need to be valid.

6.1.2 Graph Coloring Problem

The goal of the graph coloring problem is to color all vertices in a graph such that no two vertices sharing the same edge have the same color. Figure 6.2 shows a graph consisting of 10 vertices that can be colored with only 3 colors without breaking the rules of the problem. The task is either to solve the problem with a predefined number of colors or to find out the least number of colors needed to solve the problem. The latter case can be divided into multiple instances of the first case.

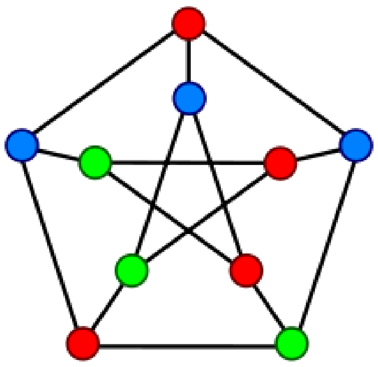


Figure 6.2: A proper graph coloring with three colors. From Wikipedia [15].

We encoded this problem to the datatype `unorderedSets`. Since the graph coloring problem is similar to how `unorderedSets` is defined, this encoding could be easily expressed with the inequality constructor. Each vertex is encoded to a variable in the `unorderedSets` datatype belonging to the set of colors available in the problem. For all vertices in the graph, each vertex should not have the same color as any of its neighbors.

6.1.3 Hamiltonian Path Problem

A Hamiltonian path in a graph is a path that passes all vertices exactly once. Figure 6.3 shows an example of a Hamiltonian path.

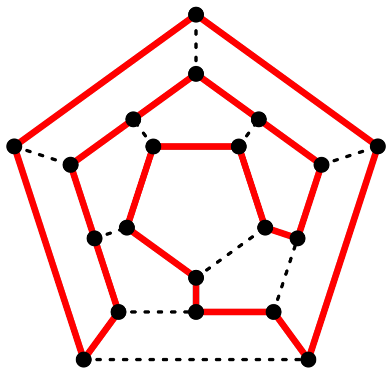


Figure 6.3: An example of a Hamiltionian path. From Wikipedia [16]. (GFDL / CC-BY-SA)

We encoded the Hamiltonian path problem to the datatype `pseudoBool`. Each edge is represented by a Boolean variable. If a vertex is visited only once, it will have at least one and at most two connecting edges traversed. Since `pseudoBool` does not include an *at most two* constraint, we need to add that logic by hand. We do this by encoding that if an edge to a vertex is traversed, at most one of the remaining connected edges will be traversed. The encoding function combines the two constraints, at least one and at most two constraints, for all edges of each vertex.

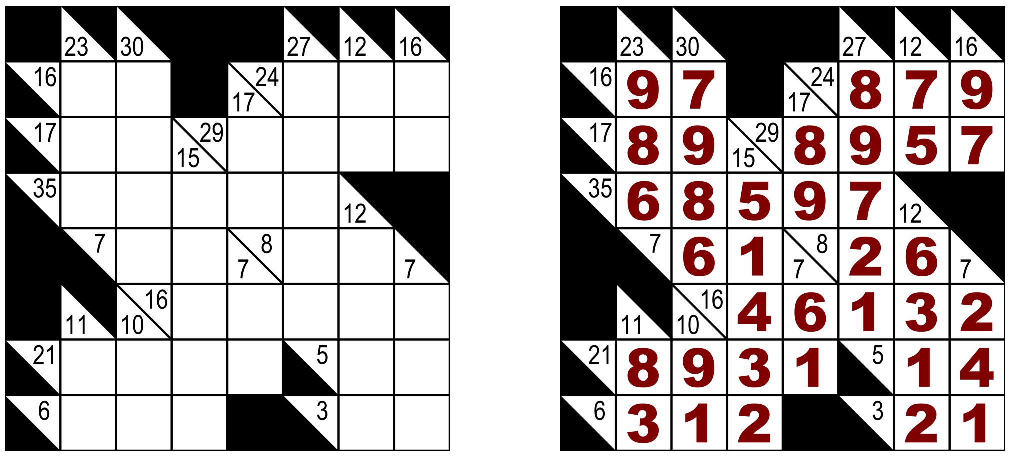
6.2 Logical Puzzles

Besides sudoku, a wide range of logical puzzles can be encoded. Below are short descriptions of the encodings of kakuro [17] and killer sudoku [18]. Other puzzles such as Dominosa[1], Filling[2], Flip[3], Magnets[4], Tents[5], Towers[6], Unequal[7], among many others (see this puzzle website[8] for more puzzles) can be encoded in similar ways.

6.2.1 Kakuro

Kakuro is a logical puzzle quite similar to sudoku. The goal is to insert digits between 1-9 into all white cells in a grid. The grid also contains black cells, called clues, specifying the sum of the white cells following to the right or below. The white cells belonging to the same clue cannot contain any duplicates. Figure 6.4 shows a kakuro problem, unsolved to the left and solved to the right.

[1]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/dominosa.html
[2]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/filling.html
[3]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/flip.html
[4]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/magnets.html
[5]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/tents.html
[6]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/towers.html
[7]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/js/unequal.html
[8]https://www.chiark.greenend.org.uk/ sgtatham/puzzles/



(a) Unsolved. From Wikipedia [19]. (GFDL / CC-BY-SA)

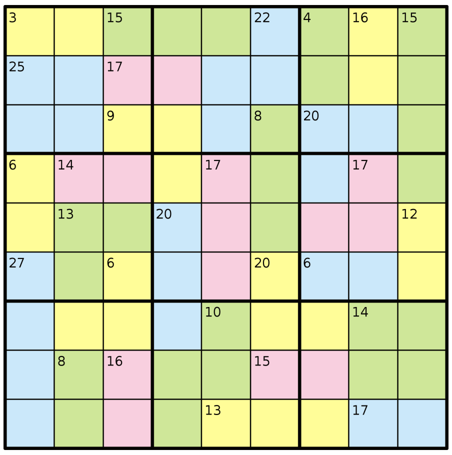
(b) Solved. From Wikipedia [20]. (GFDL / CC-BY-SA)

Figure 6.4: Example Kakuro

We encoded this problem to the datatype `numBoolRange`. Each cell is represented by a variable that ranges from 1 to 9. To calculate the sum of each clue, addition is used. Since `numBoolRange` only allows addition of two variables, the encoding will have to save the result of one addition into a new variable used in the next addition. When all additions for a clue are connected, they should equal the number given by the clue. Lastly, we add the constraint that the cells belonging to the same clue needs to be unique. This is done by encoding that none of the cells should equal another cell in the same clue.

6.2.2 Killer Sudoku

Killer sudoku is a combination of sudoku and kakuro. Like sudoku, it consists of a 9 x 9 grid with 3 x 3 blocks, and the same constraints apply to this puzzle. Killer sudoku also has *cages*, similar to the clues in kakuro. Each cage consists of several cells whose sum should equal the number defined for the cage. The numbers in a cage need to be unique. In Figure 6.5, the cages are seen in different colors.


(a) Unsolved. From Wikipedia [21].

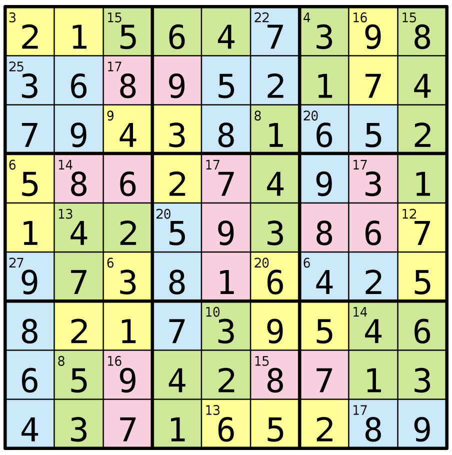

(b) Solved. From Wikipedia [22].

Figure 6.5: Example Killer Sudoku

We encoded killer sudoku to the datatype `numBoolRange` by combining the sudoku and kakuro encodings. Each cell is represented as a variable that ranges from 1 to 9, as both puzzles have in common. All rows, columns, and blocks are encoded as a regular sudoku, and the cages are encoded in the same way as the clues in kakuro.

7
Discussion

This project has implemented verified functions for encoding problems to SAT. The final products are three different datatypes; one for Boolean expressions, one for natural numbers, and one for sets of other types. The encoding functions from these datatypes to CNF are encoded and proved correct in HOL4 and compiled using CakeML.

The focus of this project has been to create *verified* encoding functions, and all encoding functions described in Chapter 4 have been proved correct. However, the focus has not been on creating efficient encoding functions. Section 7.2 discusses how our encodings compare to other encodings.

Chapter 6 shows a range of problems and how they are encoded to one of the three main datatypes. However, there are still problem types that the datatypes do not cover. Section 7.4 discusses how the work can be extended to include more problem types.

7.1 Contribution to CakeML

This project has been implemented as part of CakeML. CakeML includes a checker for SAT solver output, *cake_lpr* [6]. This software can, together with a SAT solver, produce a verified solution to a SAT problem.

If cake_lpr is combined with the encoding chain implemented in this project, more problems than those in CNF can be solved. This opens up opportunities for broader usage of CakeML on verified problem solving.

7.2 Related Work

In 2010, Anuarul Hoque et al. [23] used HOL to verify encodings for specific problems. This verification technique individually verified the encoding of *each problem* instead of verifying the encoding from one datatype to another. This resulted in an added verification time for each problem and a need to write a new proof for each problem. The encoding presented in this report instead covers all verification in the implementation in order for easier usage.

Proving the general encoding process is not yet a common research topic, but an example can be found in a paper by Ishii et al. in 2020 [24]. They showed how it is possible to verify two encoding methods called k-induction [25] and IC3/PDR [26]. This was done with the Coq proof assistant [27]. They think they are the first to formally verify a tool for these encoding methods.

Since this is a relatively new research subject, we believe we are the first to write a verified SAT-encoding tool in HOL4.

7.2.1 SMT Solvers and QBF Solvers

SMT (satisfiability modulo theories) solvers and QBF (quantified Boolean formula) solvers are two problem solvers similar to SAT solvers but with different input specialties. SMT solvers are good at arithmetics and can solve equations similar to the datatype for natural numbers developed in this project [28]. QBF solvers solve quantified Boolean formulas, i.e., formulas with the quantifiers $\forall$ and $\exists$ over Boolean variables [29].

SMT and QBF solvers can be used in place of SAT solvers, and for many problems they are better suited. However, they do not produce a proof trace in the same way as SAT solvers do. This means that the correctness is harder to verify. In comparison, this project provides a verified way of solving similar problems.

7.3 Encoding Strategies

The encoding from the datatype `boolExp` to `cnf` was done in two different ways. The first way used a naive encoding approach with de Morgan's law and the distribution property. This could potentially lead to an exponential increase in equation size. The second way used Tseytin transformation [11] to get a linear growth relative to the input size. It is not always the case that encodings using Tseytins transformation are smaller or more efficient than when using de Morgan's law and the distribution property.

The reason for doing the encoding in two different ways was that the naive encoding from `boolExp` to `cnf` turned out to be a bottleneck in the encoding. Encoding a sudoku with the naive approach, for example, lead to a CNF expression bigger than CakeML could handle. Using Tseytin transformation, there was no problem encoding a sudoku. Even though using Tseytin transformation is not more efficient than the naive encoding in all cases, we think the improvement in cases like the sudoku is motivation enough to make the change to use Tseytin transformation in our encoding function.

However, Tseytin transformation involves creating new variables. Theoretically, this should not be a problem, but in practice, it made things more difficult. Firstly, the encoding process needs to keep track of all new variables and what they should be used for. Secondly, the assignment function w that assigns Boolean values to Boolean variables, also needs to assign the new variables correctly. Both of these things also make the proofs of preservation of satisfiability more difficult.

The focus of this project has been on making the encodings as simple as possible, however with the encoding from `boolExp` to `cnf` the simplest solution was not enough.

Further down the encoding chain is the encoding of the pseudo-Boolean constraints at least one, at most one, and exactly one. This encoding process can also be improved by constructing new variables, using a technique called *Commander-Variable encoding* [30]. This encoding method is one of many improvements to the encoding of pseudo-Boolean constraints [31]. Commander-variable encoding has been shown to be faster than the naive encoding [30].

The implementation of the Commander-Variable encoding in this project would again lead to all the difficulties with introducing new variables, which is one of the reasons why this is not done. Another reason is that the encoding of pseudo-Boolean constraints never turned out to be a bottleneck, and implementing and improving other parts of the encoding chain was prioritized.

The encoding of natural numbers was not done using the most naive encoding *direct encoding* but instead *order encoding*. In direct encoding, the Boolean variables represent $x = i$ for some number variable x and some constant i, as opposed to order encoding where the Boolean variables represent $x \leqslant i$.

Tamura et al. [32] showed that order encoding performs better than direct encoding and a third encoding method called support encoding on a set of benchmarking problems. Order encoding was able to both solve problems faster and solve some previously unsolved problems. Still, there exist even better encoding techniques for natural numbers than order encoding [33], which were not used in this project due to the focus not being on efficiency.

7.4 Future work

There are two different areas within which future work could be done. Firstly, the efficiency of the current encodings could be improved. Several ways of doing this were discussed in Section 7.3. Secondly, the project could be expanded with more datatypes in order to cover more problems.

The first functionality to add when expanding the encoding chain is the possibility to calculate sums of natural numbers. In the current state, our datatype for natural numbers includes addition of two numbers. However, in order to add more numbers one needs to manually construct new variables to hold the intermediate calculations. For example, $a + b + c = d$ must be encoded as $a + b = x \wedge x + c = d$. Creating a new datatype with an added sum constructor would simplify the encoding of problems such as killer sudoku and kakuro.

Some functionalities next in line after addition are the possibilities of expressing negative numbers, multiplication and division with numbers, MAX/MIN functions, and time representation such as an UNTIL operator. After these functionalities are added, additional problems can be encoded to the verified encoding chain such as the knapsack problem, scheduling problems, and nondeterministic finite automata.

7.5 Lessons learned

During the project, we learned how important it is to make the encoding functions as simple as possible. The reason for this is that it makes it easier to prove that the encoding preserves satisfiability. The choice of which predefined HOL4 functions to use is also important, since some functions have more theorems built around them than others. Furthermore, some functions are not defined for all possible inputs.

Another thing we learned is that more functionality in a datatype does not always make the proof more difficult. A concrete example of this is the encoding of natural numbers compared to the encoding of unordered sets. Verifying the encoding of unordered sets to a Boolean datatype turned out to be more difficult than verifying the encoding of natural numbers to a Boolean datatype, even though the datatype for natural numbers contains more functionality. There are several possible reasons for this, for example, that order encoding was not possible to use with unordered sets, or that in `numBool` all variables were restricted to range between 0 and some k. When the datatype for natural numbers was implemented, it could be used as the basis for the encoding of unordered sets.

One more thing we improved on over time was to divide the proofs into several lemmas that handle different layers of the proof. For example, a proof about numbers in a list might be possible to divide into one general property about numbers and one property about lists.

7.6 Conclusion

This project shows that it is possible to write verified SAT encoding functions to encode different problems to SAT. Furthermore, the case studies show that the technique is usable on real problems. The contribution of this project is a verified encoding chain that can easily be extended to new problems, reducing the number of proofs needed to get the whole encoding to CNF verified.

The development of SAT solvers is an ongoing research area, and SAT solvers are used to solve increasingly computationally heavy problems. To utilize the development of SAT solvers to solve new problems, the encoding process is crucial. Using CakeML and HOL4, one is confident that the encoding is correct, all the way down to machine code. We hope this project inspires SAT encoding research to see the potential in formal verification.

Bibliography

[1] *SAT Competitions*. The International SAT Competition Web Page. URL: `http://www.satcompetition.org/` (visited on 30/11/2020).

[2] H. H. Hoos and T. Stützle. 'SATLIB: An Online Resource for Research on SAT'. In: *Sat* (2000), pp. 283–292.

[3] M. Sheeran, S. Singh and G. Stålmarck. 'Checking Safety Properties Using Induction and a SAT-Solver'. In: *International Conference on Formal Methods in Computer-Aided Design*. Springer. 2000, pp. 127–144.

[4] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. USA: Cambridge University Press, 2004.

[5] *CakeML, a Verified Implementation of ML*. URL: `https://cakeml.org/` (visited on 27/11/2020).

[6] Y. K. Tan, M. J. H. Heule and M. O. Myreen. 'Cake_lpr: Verified Propagation Redundancy Checking in CakeML'. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2021.

[7] L. Zhang and S. Malik. 'Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications'. In: *Automation and Test in Europe Conference and Exhibition 2003 Design*. Mar. 2003, pp. 880–885.

[8] E. Goldberg and Y. Novikov. 'Verification of Proofs of Unsatisfiability for CNF Formulas'. In: *Automation and Test in Europe Conference and Exhibition 2003 Design*. Mar. 2003, pp. 886–891.

[9] *HOL, Interactive Theorem Prover*. URL: `https://hol-theorem-prover.org/` (visited on 27/11/2020).

[10] K. Slind and M. Norrish. ‘A Brief Overview of HOL4’. In: *Theorem Proving in Higher Order Logics*. Ed. by O. A. Mohamed, C. Muñoz and S. Tahar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 28–32.

[11] G. S. Tseitin. ‘On the Complexity of Derivation in Propositional Calculus’. In: *Structures in Constructive Mathematics and Mathematical Logic* (1968), pp. 115–125.

[12] N. Tamura et al. ‘Compiling Finite Linear CSP into SAT’. In: *Constraints* 14.2 (June 2009), pp. 254–272.

[13] *N-Queens Benchmark - OpenBenchmarking.Org*. URL: `https://openbenchmarking.org/test/pts/n-queens` (visited on 20/04/2021).

[14] A. Niewiadomski et al. ‘Applying Modern SAT-Solvers to Solving Hard Problems’. In: *Fundamenta Informaticae* 165.3-4 (14th Mar. 2019). Ed. by W. Penczek, H. Schlingloff and P. Wasilewski, pp. 321–344.

[15] *File:Petersen Graph 3-Coloring.Svg - Wikimedia Commons*. URL: `https://commons.wikimedia.org/wiki/File:Petersen_graph_3-coloring.svg` (visited on 18/05/2021).

[16] C. Sommer. *Hamiltonian Path through a Dodecahedron*. 27th Feb. 2007. URL: `https://commons.wikimedia.org/wiki/File:Hamiltonian_path.svg` (visited on 12/05/2021).

[17] *Kakuro*. In: *Wikipedia*. 22nd Apr. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Kakuro&oldid=1019222284` (visited on 27/04/2021).

[18] *Killer Sudoku*. In: *Wikipedia*. 10th Mar. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Killer_sudoku&oldid=1011314386` (visited on 10/05/2021).

[19] Octahedron80. *A Simple Kakuro Puzzle*. 25th Oct. 2007. URL: `https://commons.wikimedia.org/wiki/File:Kakuro_black_box.svg` (visited on 12/05/2021).

[20] Octahedron80. *A Simple Kakuro Puzzle with Solution*. 25th Oct. 2007. URL: `https://commons.wikimedia.org/wiki/File:Kakuro_black_box_solution.svg` (visited on 12/05/2021).

[21] *File:Killersudoku Color Solution.Svg*. In: *Wikipedia*. URL: `https://en.wikipedia.org/wiki/File:Killersudoku_color_solution.svg` (visited on 18/05/2021).

[22] *File:Killersudoku Color.Svg*. In: *Wikipedia*. URL: `https://en.wikipedia.org/wiki/File:Killersudoku_color.svg` (visited on 18/05/2021).

[23] K. Anuarul Hoque et al. ‘An Automated SAT Encoding-Verification Approach for Efficient Model Checking’. In: *2010 International Conference on Microelectronics*. 2010 International Conference on Microelectronics. Dec. 2010, pp. 419–422.

[24] D. Ishii and S. Fujii. ‘Formalizing the Soundness of the Encoding Methods of SAT-Based Model Checking’. In: *arXiv* (24th June 2020).

[25] A. F. Donaldson et al. ‘Software Verification Using K-Induction’. In: *Static Analysis*. Ed. by E. Yahav. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 351–368.

[26] A. R. Bradley. ‘SAT-Based Model Checking without Unrolling’. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by R. Jhala and D. Schmidt. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 70–87.

[27] *The Coq Proof Assistant*. URL: `https://coq.inria.fr/` (visited on 26/04/2021).

[28] L. de Moura and N. Bjørner. ‘Z3: An Efficient SMT Solver’. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and J. Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[29] E. Giunchiglia, M. Narizzano and A. Tacchella. ‘QuBE++: An Efficient QBF Solver’. In: *Formal Methods in Computer-Aided Design*. Ed. by A. J. Hu and A. K. Martin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 201–213.

[30] W. Klieber and G. Kwon. ‘Efficient CNF Encoding for Selecting 1 from N Objects’. In: *Proc. International Workshop on Constraints in Formal Verification* (2007), p. 14.

[31] V.-H. Nguyen et al. ‘Empirical Study on SAT-Encodings of the At-Most-One Constraint’. In: International Conference on Smart Media and Applications (SMA). 2020, p. 6.

[32] N. Tamura, T. Tanjo and M. Banbara. ‘Solving Constraint Satisfaction Problems with SAT Technology’. In: *Functional and Logic Programming*. Ed. by M. Blume, N. Kobayashi and G. Vidal. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 19–23.

[33] C. Vasconcellos-Gaete, V. Barichard and F. Lardeux. ‘Abacus: A New Hybrid Encoding for SAT Problems’. In: 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI). Nov. 2020, pp. 145–152.