Sudoku

CO3015
Computer Science Project

Final Report

submitted to the University of Leicester
in Partial Fulfilment for the degree of Bachelor of Science

Fania Raczinski

May 2007

Department of Computer Science
University of Leicester

Contents

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this, amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Fania Raczinski

Signed:

Date: 10. May 2007

Abstract

Sudoku is a logic based puzzle becoming more and more popular all over the world. Depending on the difficulty of the puzzle it can take between 10 minutes up to an hour to solve one of them. This asks for a computer aided solver to speed up this process. The aim of this project is to develop a program that can achieve this. There are two approaches I have taken. One is a Brute Force Solver written purely in the functional programming language Haskell and the other is to translate the Sudoku problem into DIMCAS CNF format such that the SAT Solver MiniSat can solve it. Having integrated both Solvers into my program the user can choose between them by pressing buttons on the GUI provided. In order to help a user solve a puzzle by himself a check button is implemented which gives feedback on individual moves or a completed puzzle.

I have shown through two examples, the Brute Force and the SAT Solver, how a program fully designed in Haskell would solve a Sudoku puzzle and I have given indications on how fast solutions are produced. MiniSat didn’t have any problems solving even harder puzzles while the Haskell Solver had major problems solving even easiest puzzles, depending on the position of the empty fields.

I have deepened my knowledge about functional programming and Haskell and gained insight into the to me totally new area of SAT.

1 Introduction

1.1 Motivation

Sudoku [1] has become increasingly popular in the past few years. The puzzle appears in newspapers and magazines, in books and scientific papers and there even exist TV Shows about it. There are many reasons for its international popularity. The rules are simple and easy and it is based on numbers instead of words or letters. Nevertheless there are quite complex mathematical ideas behind it and solving such a puzzle can prove quite difficult and time consuming.

The idea of this project was to create a computer program which will aid in solving a Sudoku puzzle. This includes some sort of graphical user interface where the user can enter puzzles. The program should offer the possibility of solving the puzzle for the user and check individual moves or finished puzzles. Two solvers should be implemented. One should be making use of a SAT [2] solver and the other should be a simple Brute Force solver.

Haskell [3] has been my favourite programming language since the day I started learning it. It's purely functional, concise, very simple and very beautiful. Therefore I want to write this project using Haskell, including the graphical user interface.

1.2 Objectives

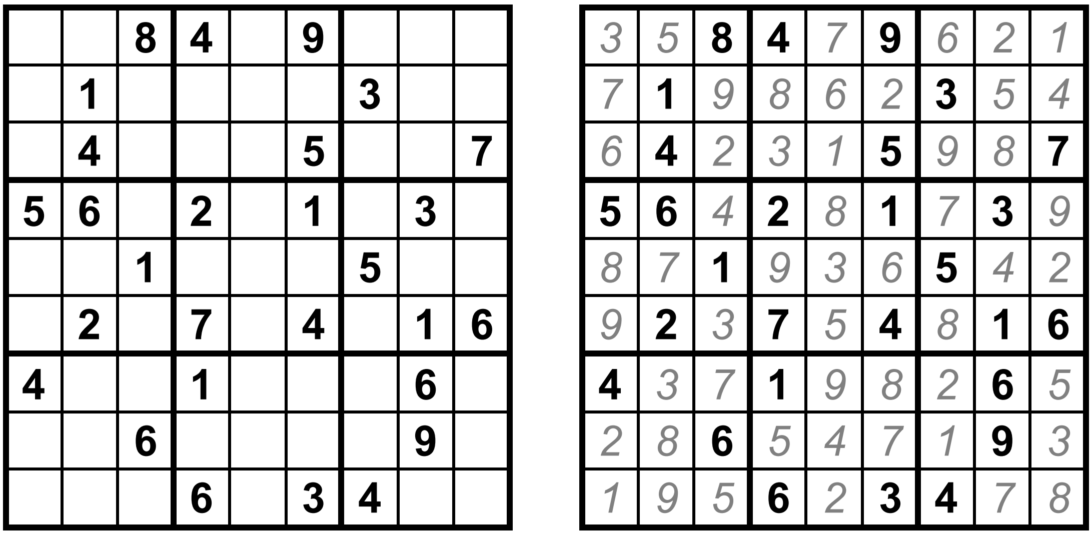
The main objectives of this project will be explained in the following list of points.

1. The graphical user interface should be written using the Haskell programming language together with the GUI library Gtk2Hs [4] and have the following features to make the game play as convenient as possible for the user.
    a. A 9x9 grid with 81 text entry fields to enter Sudoku puzzles by hand or load example puzzles.
    b. A button to check whether individual moves or a completed grid is correct.
    c. A button to let a brute force solver solve the puzzle.
    d. A button to let a SAT Solver solve the puzzle.
    e. A button to clear the grid.
    f. A label which returns some feedback to the user each time any of the buttons is pressed.
2. The Brute Force Solver should be written purely in Haskell and not be based on any complex heuristics. The purpose of this solver is to show how long it can take the computer to calculate the solution to such a problem.
3. The SAT Solver MiniSat [5] should be integrated into the program.
4. A conversion program needs to be written in order to convert a given Sudoku puzzle into the DIMACS CNF format [6], which is the necessary input format for MiniSat. This conversion program also needs to be able to convert the output file produced by the SAT Solver back into some kind of form such that it can be displayed on the GUI for the user.

2 Literature Survey

2.1 Sudoku

“Sudoku (数独 *sūdoku*), also known as Number Place or Nanpure, is a logic-based placement puzzle. The aim of the puzzle is to enter the digits 1 through 9 in each cell of a 9×9 grid made up of 3×3 sub grids (called "regions") so that each row, column, and region contains exactly one instance of each digit. A set of clues, or "givens", constrain the puzzle such that there is only one way to correctly fill in the remainder.” [1]



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

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

Fig.1. Sudoku Puzzle and Solution

Figure 1 shows an example Sudoku. On the left is the empty grid with 28 given numbers out of a standard size puzzle with 81 cells. To the right is the solution to this specific instance.

2.1.1 Background

The puzzle as we know it today was invented by Howard Garns in 1979 and published in Dell Magazines under the name of Number Place. In 1986 it became popular in Japan when the publisher Nikoli [7] rediscovered it from old Dell publications. Sudoku is actually a special case of Latin Squares [8] with the additional constraint for each sub grid. Nowadays there exist several variations of the game. These vary from different sizes, different shapes and different rules to the integration of colours into the game. Examples are Comparison Sudoku, Samurai Sudoku or Killer Sudoku. The fact that this puzzle uses numbers is actually irrelevant. Any set of distinct symbols would work fine as well, such as letters or small pictures. The name “Sudoku” itself is an abbreviation of a Japanese sentence (数字は独身に限る "*Sūji wa dokushin ni kagiru*" - "the digits must occur only once") and is a trademark of the Japanese publisher Nikoli Co. Ltd.

2.1.2 Facts

- The minimum amount of given numbers in a Sudoku needed to yield a unique solution is still an unsolved problem. The minimum number known, but not proven, is 17 [9] though.
- The maximum number of given numbers in a Sudoku that still does not yield a unique solution, is 77, so four empty fields.
- The number of possible solutions for one single empty row alone is 9! = 362880.
- Disregarding all rules and constraints there are 9^{81} different combinations of a 9x9 grid. (1.9662705047555291361807590852691e+77)
- The number of possible Sudoku solutions to an empty 9x9 grid is 6,670,903,752,021,072,936,960 (calculated by Bertram Felgenhauer [10])
- There is an NP-complete [11] problem related to Sudoku: Given a partially completed grid, determine whether it has a solution.

2.1.3 Solving Techniques

Although each traditional Sudoku has only one solution there are several ways to find it. Wikipedia [12] suggests that there are three processes to solve a puzzle. Scanning, marking up and analyzing. These techniques can be in any order and repeated if necessary. Scanning and analysing consist of several different techniques to identify possible solutions, or eliminate invalid solutions, to each field in the grid. They take into account the constraints given and any contingencies that might arise. Marking up is very useful for solving puzzles of higher difficulty. It consists of either making small notes of which numbers could be in the relevant cell of the grid or which numbers cannot be in it and can be done using subscripts or small dots.

Computers, of course, use slightly different approaches. They are capable of more advanced methods but nevertheless, even a computer could still need an unreasonable amount of time to solve such a puzzle. There are many techniques, heuristics that can speed up that process immensely. Explaining even some of them is a whole paper on its own though and I won’t even try to enumerate them. Google is going to be of more help than I ever could be in this case. Search for “Sudoku solving techniques” or “Sudoku heuristics”.

2.1.4 Sources

- **Wikipedia about Sudoku** [1] – Wikipedia discusses the origins of Sudoku as well as several solving techniques or the Mathematics behind it. It is a very useful side, summarising everything we need to know about the game.
- **Nikoli Webpage** [7] – Nikolis official webpage with some nice and simple explanations about the game.

2.2 SAT

2.2.1 Boolean Satisfiability Problem

SAT actually stands for the Boolean satisfiability problem or simply the Satisfiability Problem and is a decision problem considered in complexity theory.

A given Boolean expression (written in propositional logic using AND □, OR □, NOT ¬, variables and parentheses only) is satisfiable if we can assign logical values to its variables such that it makes the formula true.

This problem was the first problem to be proven NP-complete by Stephen Cook in 1971 [13]. A problem is in the NP class (nondeterministic polynomial time) if it is solvable in polynomial time by a nondeterministic Turing machine. NP-complete means it belongs to the hardest problems to solve efficiently in NP.

We can restrict the problem somewhat if we use sentences that are conjunctions of clauses where each clause contains at most three literals. This is called 3-SAT and is still NP-complete. We can restrict this further by using only sentences where each clause contains at most 2 literals. We call this 2-SAT and interestingly this is in P.

2.2.2 CNF

3-SAT is expressed in conjunctive normal form (CNF) [14]. A formula is a conjunction (□) of clauses and each clause is a disjunction (□) of possibly negated (¬) literals (see Figure 2 and 3).

$$(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$$

Fig.2. CNF Example

$$A \wedge (B \vee (D \wedge E)). \rightarrow A \wedge (B \vee D) \wedge (B \vee E).$$

Fig.3. Converting a formula into CNF

A satisfiability problem remains NP-complete even if written in CNF. 2-satisfiabilty (2-SAT) for example limits the number of literals in a clause to 2, is written in disjunctive normal form and can be solved in polynomial time. But this is not really relevant to the problem of this project. CNF is important to keep in mind though and I will explain why in chapter 4.1.4.

2.2.3 SAT Solvers

SAT Solvers use propositional formulas in conjunctive normal form. In order to use the SAT Solver to find a solution to a Sudoku we need to transform the puzzle into logical formulae in DIMACS CNF format [6]. This is the current benchmark for SAT competitions [15] and most SAT Solvers including MiniSat which I will be using. The SAT Solver will then try to find the necessary assignments to all literals in the formula such that the sentence becomes true and returns them. This result then needs to be converted back into a Sudoku problem and displayed for the user.

Most modern SAT Solvers are variations of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [16]. This algorithm is based on backtracking search procedures for deciding the satisfiability of propositional logic formulae in conjunctive normal form. The variations to it include among others conflict analysis, clause learning, non-chronological backtracking and random restarts.

2.2.3 Sources

- **Wikipedia on Boolean satisfiability problem** [2]
- **Wikipedia on Conjunctive normal form** [14]
- **Wikipedia on NP-complete** [11]
- **SAT Competitions** [15] - The webpage for the SAT Solver where I found MiniSat.
- **The Sudoku Puzzle as a Satisfiability Problem** [17] - This is a webpage with an embedded working Sudoku Solver implemented using the Java Library SAT4J [18]. **The SAT Game** [19] – This is a very playful and interesting page about SAT. Learning and understanding by doing.

2.3 Haskell

Haskell is the most popular purely functional, lazy programming language. Some of its features are static typing, higher order functions, polymorphism, type classes, and monadic effects. The main difference to imperative languages is that they consist of a sequence of commands, which are executed one after the other and a Haskell program is a single expression, which is executed by evaluating the expression when needed. More information about Haskell can be found at any of the following sources.

2.3.1 Sources

- **Haskell.org** [3] – The main webpage for the Haskell community.
- **Wikibooks Haskell** [20] – The Haskell Wiki Book. Very clear and useful.
- **Hoogle** [21] - Google for Haskell!! Very useful ☺
- **Haskell: The Craft of Function Programming**, Simon Thompson [22]
- **The Haskell School of Expression**, Paul Hudak [23]
- **Haskell Sudoku Solvers** [24] - A nice collection of several Sudoku Solvers written in Haskell.

3 Planning and Timescale

I will reproduce the original timescale and planning details of my interim report and outline the changes in chapter 3.2.1.

As I already stated in the Project Plan, there are several tasks I need to tackle throughout the course of the year and there will not be a clear separation between Semester 1 and 2. Nevertheless, my original idea was to follow the Rational Unified Process [25] and go through its four phases; Inception, Elaboration, Construction and Transition. Due to a lot of delays during Semester 1, this original plan to finish the first two phases (Inception and Elaboration) by now could not be fulfilled. Reason for this delay was most of all the fact that the programming language was decided upon so late into the Semester. If I had known from the start which language to use, I could have prepared better and I could have saved a lot of time just revising and studying Haskell. I will compensate for this by finishing most of the work originally planned for Semester 1 during the winter vacation.

3.1 Semester 1

The planning for Semester 1, see figure 5 below, has been slightly changed due to one main reason. After long thought, I decided to write my project in Haskell. I like functional programming and prefer using and extending my knowledge on this rather than Java or C++. I also find it more challenging to use a functional programming language and teach myself how to write GUIs and how to properly handle user input / output.

As this is a functional programming language though, it is much more difficult to come up with a useful set of requirements and a specification. The implementation of a program in Haskell is after all very different to a program written in any other object oriented programming language. Class diagrams don't apply at all for example since there are no classes in Haskell. There is still much thought to be done about this issue. The original timescale as given in figure 4 has been affected by these changes as well.

2006											
	October					November				December	
Week	1	2	3	4	5	6	7	8	9	10	11
Requirements			■	■	■						
Specification			■	■	■						
Interface Prototype				■	■	■	■	■			
Program Design				■	■	■	■	■			
Brute Force Prototype				■	■	■	■	■	■		
CNF Encoding								■	■	■	■
Start Date	02.10	09.10	16.10	23.10	30.10	06.11	13.11	20.11	27.11	04.12	11.12

Fig.4. Ghantt Chart for Semester 1

3.1.1 Changes for Semester 1

To illustrate the changes for Semester 1, I have composed the following list. I have also changed figure 5 and replaced the Challenge/Problem column by a Progress column to indicate what has been completed and what not.

Requirements

I delayed writing the requirements so far because I wasn't sure how to write requirements for a Haskell program or if they are needed and useful at all in this case. I will think about this before start of Semester 2.

Specification

It was too unclear up to now how the system will be implemented. This will be done in preparation for Semester 2.

Interface Design (Prototype)

This has been done and will be further extended and build upon during Semester 2.

Program Design

This will be done together with the Brute Force Implementation during the vacation.

Brute Force Attempt (Prototype)

I have not started the Brute Force attempt yet because I wanted to finish the prototype for the interface first. This has been achieved now so I can implement the Brute Force Solver in preparation for Semester 2.

CNF Encoding of Sudoku

I have done more reading on this but since the Brute Force Attempt has been delayed so much of course it would not make sense to start working of this encoding problem now.

Task	Description	Progress
Requirements	Compose a set of structured requirements fit for this project.	Delayed.
Specification	Detail the specifications for the system to be build.	Delayed.
Interface Design (Prototype)	Design and implement an interface for the game (Prototype).	Completed.
Program Design	Design a rough idea of the program.	Delayed./ In progress
Brute Force Attempt (Prototype)	Try a brute force approach to the solver.	Not started.
CNF Encoding of Sudoku	Translate the Sudoku problem into DIMACS CNF format.	Not started.

Fig.5. Plan for Semester 1

3.2 Semester 2

The overall plan for Semester 2 has not changed much apart from the few things that will need to be finished during the first couple of weeks after the start of term. In general of course I want to aim for having the remaining parts from Semester 1 finished by January but to make sure I will calculate in some time for at least the CNF Encoding during Semester 2 as well. This will be reflected in both figure 6 and 7.

Task	Description	Challenge/Problem
CNF Encoding	Find out how to convert a Sudoku problem into CNF format to input to the SAT Solver.	Needs to be done before I can start with the SAT Solver otherwise I am stuck.
SAT Solver Integration (Prototype)	Integrate the SAT Solver into existing code of program.	Needs to be integrated correctly and the CNF input must be correct otherwise the Solver won't produce a result.
Interface Implementation	Change and finish the program interface.	Depending on programming language to be used this might prove tricky.
Theory	Write up the theoretical background on the project.	Composing a sensible summary of all theories can be time consuming.
Technical Info	Write up technical information on the project.	Explaining all the technical detail can be time consuming and difficult.
Software Tests	Conduct software tests.	Errors or bugs at this stage will be difficult to correct.
Final Software	Finish the final software product.	Might be very time consuming.
Final Report	Write up final report.	Will be very time consuming.

Fig.6. Plan for Semester 2

	2007															
	January		February				March				April					May
Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CNF Encoding	■	■	■								Easter Vacation					Deadline 10th May!
SAT Solver Integration (Prototype)	■	■	■	■	■	■	■	■	■	■						
Interface Implementation	■	■	■	■	■	■	■	■	■	■						
Theory								■	■	■						
Technical Info								■	■	■						
Software Tests					■	■	■	■	■	■						
Final Software								■	■	■						
Final Report								■	■	■						▨
Start Date	22.01	29.01	05.02	12.02	19.02	26.02	05.03	12.03	19.03	26.03	02.04	09.04	16.04	23.04	30.04	07.05

Fig.7. Ghantt Chart for Semester 2

3.2.1 Changes for Semester 2

There have only been slight time changes in this semester so I will just show a new version of the Ghantt Chart for Semester 2 in figure 8.

	2007															
	January		February				March				April					May
Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CNF Encoding									■	■	Easter Vacation					Deadline 10th May!
SAT Solver Integration (Prototype)									■	■						
Interface Implementation									■	■						
Theory			■	■	■	■	■	■	■	■						
Technical Info								■	■	■						
Software Tests			■	■	■	■	■	■	■	■						
Final Software			■	■	■	■	■	■	■	■						
Final Report																▨
Start Date	22.01	29.01	05.02	12.02	19.02	26.02	05.03	12.03	19.03	26.03	02.04	09.04	16.04	23.04	30.04	07.05

Fig.8. Revised Ghantt Chart

Coding the Brute Force Solver has taken up much more time than originally intended. The language difficulties proved a bigger problem and especially trying to improve the graphical user interface took up a lot of time. This had its impact on all the rest of the project. Implementing the SAT Solver and developing the conversion program for the DIMACS CNF format has been delayed by several weeks. The writing and most of the coding for the SAT Solver integration / CNF encoding has been done during the Easter vacation.

4 Project Core

4.1 Technical Background Information

I have provided all necessary software needed to run my program on the CD submitted with this report. This includes the Haskell compiler, the GUI library and also in addition the MiniSat source just for completeness. Of course I have included a separate readme file as well which explains in detail how to run my program or how to install Haskell and the GUI library. I am using Cygwin here on my Windows machine, but it is not necessary. I used it to compile the Haskell files into an executable and also to test run the SAT Solver independently from my program. I used GHCi to test all my Haskell functions. GHCi is an interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreted.

4.1.1 Haskell

According to my original plan I managed to use Haskell to write all my code. I thought it would be much more challenging, interesting and also rewarding for me to use a functional programming language as opposed to an object-oriented language like Java.

Since some of the modules I have written contain many complex functions that require detailed explanations, I have chosen to make use of literate Haskell files (.lhs) next to normal Haskell files (.hs). The main difference is simply that everything in such a literate file will be considered a comment unless the line starts with a “>” symbol. This is very useful if many comments need to be written inside the code and the file can still be compiled and run just as any normal Haskell file.

I am using the Glasgow Haskell Compiler (GHC [26]). In order to work with the GUI Library for Haskell I need to use version 6.6 of the compiler.

4.1.2 GUI Library

As I have mentioned before, to build graphical user interfaces Haskell needs a separate GUI Library. After a lot of research I found the Gtk2Hs GUI library [4] to be the most suitable. Gtk2Hs supports the usage of Glade [27] which is a Graphical User Interface builder that generates XML files which can then be accessed and used in my Haskell code. I will explain this procedure in more detail later on though. I am using the most recent version of this library, version 0.9.11.

4.1.3 SAT Solver

For the SAT Solver I have done much research and I found that MiniSat, designed by Niklas Eén and Niklas Sörensson, seems the most suitable. On their website [5] they claim that "MiniSat is a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT. Together with SatELite, MiniSat was recently awarded in the three industrial categories and one of the "crafted" categories of the SAT 2005 competition." I am using version 1.14 of the Solver which is the latest version available before the release of MiniSat2 in the end of 2006.

The use of MiniSat itself is very simple. After it is compiled we can execute it with two arguments. The input file and the output file. MiniSat takes input files in DIMACS CNF format. Running the Solver on a problem also prints out some general information. An example for this when resulting in a satisfiable problem is this:

```
============================[MINISAT]===============================
|Conflicts|     ORIGINAL    |            LEARNT           |Progress|
|         |Clauses Literals| Limit Clauses Literals Lit/Cl |        |
====================================================================
|       0 |  2065    23481 |  688      0        0     nan   | 0.000 %|
====================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 271            (8742 /sec)
propagations          : 999            (32226 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 42.25 MB
CPU time              : 0.031 s

SATISFIABLE
```

The actual output file for this example is shown below. The first line says that is it satisfiable and is followed by one clause specifying which variables are valid and which aren't. For demonstration purposes I have highlighted them. These indicate the values for each field in the solved Sudoku puzzle.

```
SAT
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -
20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -
37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47 -48 -49 -50 -51 -52 -53 -
54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -
71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -
88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98 -99 -100 -101 -102 -103 -
104 -105 -106 -107 -108 -109 -110 111 -112 -113 -114 -115 -116 -117 -
118 -119 -120 -121 -122 -123 -124 -125 126 -127 -128 -129 -130 -131 -
132 -133 -134 -135 -136 -137 -138 139 -140 -141 142 -143 -144 -145 -
146 -147 -148 -149 -150 -151 -152 -153 -154 -155 -156 -157 158 -159 -
160 -161 -162 -163 -164 -165 -166 167 -168 -169 -170 -171 -172 173 -
174 -175 -176 -177 -178 -179 -180 -181 -182 -183 184 -185 -186 -187 -
188 -189 -190 -191 -192 -193 -194 195 -196 -197 -198 -199 -200 -201 -
202 -203 -204 -205 -206 -207 -208 -209 -210 -211 -212 -213 -214 -215
-216 217 -218 -219 -220 -221 222 -223 -224 -225 -226 -227 -228 -229 -
230 -231 -232 -233 -234 235 -236 -237 -238 -239 -240 -241 -242 -243 -
244 -245 -246 -247 -248 249 -250 -251 -252 253 -254 -255 -256 -257 -
258 -259 -260 -261 -262 -263 264 -265 -266 -267 -268 -269 -270 -271 -
272 -273 -274 -275 276 -277 -278 -279 -280 -281 -282 -283 -284 -285 -
286 -287 288 -289 -290 291 -292 -293 -294 -295 -296 -297 -298 -299 -
```

```
300 -301 -302 -303 -304 -305 -306 -307 -308 -309 -310 -311 -312 -313
314 -315 -316 -317 -318 -319 -320 -321 -322 323 -324 -325 -326 -327 -
328 -329 -330 -331 -332 -333 -334 -335 -336 -337 338 -339 -340 -341 -
342 -343 -344 -345 346 -347 -348 -349 -350 351 -352 -353 -354 -355 -
356 -357 -358 -359 -360 -361 -362 -363 -364 365 -366 -367 -368 -369 -
370 -371 -372 -373 -374 -375 -376 -377 -378 379 -380 -381 382 -383 -
384 -385 -386 -387 -388 -389 -390 -391 -392 -393 -394 -395 -396 397 -
398 -399 -400 -401 -402 -403 -404 -405 -406 -407 -408 -409 -410 -411
-412 -413 -414 415 -416 -417 -418 -419 -420 -421 -422 -423 -424 -425
-426 -427 428 -429 -430 -431 432 -433 -434 -435 -436 -437 -438 -439 -
440 -441 -442 -443 444 -445 -446 -447 -448 -449 -450 -451 -452 -453 -
454 -455 456 -457 -458 -459 -460 461 -462 -463 -464 -465 -466 -467 -
468 -469 -470 -471 -472 -473 -474 -475 -476 477 -478 -479 -480 -481 -
482 483 -484 -485 -486 -487 -488 -489 -490 -491 -492 -493 -494 -495 -
496 -497 -498 499 -500 -501 -502 -503 -504 -505 -506 -507 -508 -509 -
510 -511 -512 -513 -514 -515 516 -517 -518 -519 -520 -521 -522 -523 -
524 -525 -526 -527 -528 529 -530 531 -532 -533 -534 -535 -536 -537 -
538 -539 -540 -541 -542 -543 -544 -545 -546 -547 548 -549 -550 -551 -
552 -553 -554 -555 -556 557 -558 -559 -560 -561 -562 563 -564 -565 -
566 -567 -568 -569 -570 -571 572 -573 -574 -575 -576 -577 -578 -579 -
580 -581 -582 -583 -584 585 -586 -587 -588 -589 -590 -591 -592 -593
594 -595 -596 -597 -598 -599 -600 -601 -602 -603 -604 -605 -606 -607
-608 -609 -610 -611 -612 613 -614 -615 -616 -617 -618 -619 -620 -621
-622 -623 -624 -625 -626 627 -628 -629 -630 -631 -632 -633 634 -635 -
636 -637 -638 -639 -640 -641 -642 -643 -644 645 -646 -647 -648 -649 -
650 -651 -652 -653 -654 -655 -656 -657 -658 659 -660 -661 662 -663 -
664 -665 -666 -667 -668 -669 -670 -671 -672 -673 -674 -675 -676 -677
678 -679 -680 681 -682 -683 -684 -685 -686 -687 -688 -689 -690 -691 -
692 -693 -694 -695 696 -697 -698 -699 -700 -701 -702 -703 -704 -705 -
706 -707 -708 -709 -710 -711 712 -713 -714 -715 -716 -717 -718 -719 -
720 721 -722 -723 -724 -725 -726 -727 -728 -729 -730 -731 -732 -733 -
734 -735 -736 737 -738 -739 -740 -741 -742 743 -744 -745 -746 -747 -
748 -749 -750 -751 -752 -753 754 -755 -756 -757 -758 -759 -760 -761 -
762 -763 -764 -765 766 -767 -768 -769 -770 -771 -772 -773 -774 775 -
776 -777 -778 -779 -780 -781 -782 -783 -784 -785 -786 -787 -788 789 -
790 -791 -792 -793 -794 -795 -796 -797 798 -799 -800 -801 -802 -803 -
804 -805 -806 -807 -808 -809 -810 -811 -812 -813 -814 -815 -816 -817
818 -819 -820 -821 -822 -823 -824 825 -826 -827 -828 -829 -830 -831 -
832 -833 -834 -835 836 -837 -838 -839 -840 841 -842 -843 -844 -845 -
846 -847 -848 -849 -850 -851 852 -853 -854 -855 -856 -857 -858 -859 -
860 -861 -862 -863 -864 -865 -866 -867 -868 869 -870 -871 -872 -873
874 -875 -876 -877 -878 -879 -880 -881 -882 -883 -884 -885 -886 887 -
888 -889 -890 -891 -892 893 -894 -895 -896 -897 -898 -899 -900 -901 -
902 -903 -904 -905 -906 -907 -908 -909 -910 -911 -912 -913 -914 -915
-916 -917 -918 919 -920 -921 -922 -923 924 -925 -926 -927 -928 -929 -
930 -931 -932 933 -934 -935 -936 -937 -938 -939 -940 -941 -942 -943 -
944 -945 -946 947 -948 -949 -950 -951 -952 -953 -954 955 -956 -957 -
958 -959 -960 -961 -962 -963 -964 -965 -966 -967 968 -969 -970 971 -
972 -973 -974 -975 -976 -977 -978 -979 -980 -981 -982 -983 -984 -985
986 -987 -988 -989 -990 -991 992 -993 -994 -995 -996 -997 -998 -999 0
```

And here is a counter example.

```
restarts              : 0
conflicts             : 0              (0 /sec)
decisions             : 0              (0 /sec)
propagations          : 44             (1419 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 42.12 MB
CPU time              : 0.031 s

UNSATISFIABLE
```

This is the corresponding output file containing just one line stating that it is unsatisfiable.

```
UNSAT
```

4.1.4 DIMACS CNF

The DIMACS CNF format [6] was created to provide a common format for the second DIMACS Challenges [28] in 1993 and has been suggested for the use of any SAT Solvers developed for the SAT Competitions [15].

There are different types of lines we can use. One is the comment line. It starts with a lower case letter “c” and the usage is completely optional. Any comments will be ignored by the SAT Solver.

```
c This is an example comment line.
```

The Problem line on the other hand is required. The format of this line is as follows.

```
p cnf number_of_variables number_of_clauses
```

After this all that’s left to do is to add the actual clauses. Each clause is represented by a sequence of numbers with the number 0 reserved as the end-of-clause-character so to speak. This allows clauses to be spread over several lines for example. The negation of a variable “x” will be “–x”. Just a reminder, as explained in chapter 2.2.2, the conjunctive normal form consists of a conjunction of any number of clauses, where each clause itself is a disjunction of any number of variables or the negations of variables. The following is an example input.cnf file for the example given in the mentioned chapter.

```
c Example from Chapter 2.2.2
c (A ∨ B) ∧ (¬B ∨ C ∨ ¬D) ∧ (D ∨ ¬E)
p cnf 5 3
1 2 0
-2 3 -4 0
4 -5 0
```

4.1.4.1 Translating Sudoku

Translating the Sudoku puzzle into this format is fairly easy. The general design of the file will be as shown in figure 9.

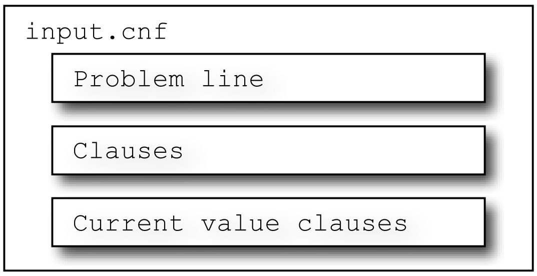
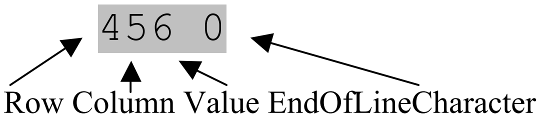


Fig.9. Design of cnf file

Since we need to know the number of clauses in order to construct the problem line, we need to find out how many clauses there will be in total before we do anything else. Let's work our way to the problem line from the bottom.

The current values of the list will be added by simply one variable for each field. The variables will consist of three digit numbers. The first digit will be the row index, the second digit will be the column index and the third digit will be the actual value inside the field.



Each line of course also needs the 0 at the end to mark the end the clause. The Sudoku puzzle in figure 1 can be translated the following way.

		138 0	144 0		169 0			
	221 0					273 0		
	324 0				365 0			397 0
415 0	426 0		442 0		461 0		483 0	
		531 0				575 0		
	622 0		647 0		664 0		681 0	696 0
714 0			741 0				786 0	
		836 0					889 0	
			946 0		963 0	974 0		

Fig.10. DIMACS CNF for current values in a grid.

The clauses to describe the Sudoku rules are a bit more complicated. Let's look over the rules of Sudoku again and summarize them.

> "The aim of the puzzle is to enter the digits 1 through 9 in each cell of a 9×9 grid made up of 3×3 sub grids (called "regions") so that each row, column, and region contains exactly one instance of each digit. A set of clues, or "givens", constrain the puzzle such that there is only one way to correctly fill in the remainder" [1]

1. Each field contains one of the numbers from 1 to 9.
2. No two fields in any cell contain the same value.
3. No two fields in any row contain the same value.
4. No two fields in any column contain the same value.

We can therefore divide the middle part into 4 blocks. One block for the individual fields, one for the cells, one for the rows and one for the columns. First of all we need enough variables to cover each field in the grid. We need to consider the different values each field can have too. So if we combine the indexes of the 81 fields in the grid (11,12,13,…,97,98,99) with the numbers from 1 to 9 then we get a list of 729 variables.

The clauses for field (1,1) are shown below. First we say that in field (1,1) we can have any values of 1 to 9. Then we give constraints saying that it can't have value 1 and 2 or value 1 and 3 etc. This forms a list of 37 clauses for one field only.

```
111 112 113 114 115 116 117 118 119 0
-111 -112 0
-111 -113 0
-111 -114 0
-111 -115 0
-111 -116 0
-111 -117 0
-111 -118 0
-111 -119 0
-112 -113 0
-112 -114 0
-112 -115 0
-112 -116 0
-112 -117 0
-112 -118 0
-112 -119 0
-113 -114 0
-113 -115 0
-113 -116 0
-113 -117 0
-113 -118 0
-113 -119 0
-114 -115 0
-114 -116 0
-114 -117 0
-114 -118 0
-114 -119 0
-115 -116 0
-115 -117 0
-115 -118 0
-115 -119 0
-116 -117 0
-116 -118 0
-116 -119 0
-117 -118 0
-117 -119 0
-118 -119 0
```

We need to specify this for each of the remaining 80 fields in the same way. All in all for the first block we will therefore need 37 clauses * 81 fields = 2997 clauses.

Similarly we need to specify the rules for each cell. The following example is the list of rules needed to say that one of the fields in the first cell could contain the value 1.

```
111 121 131 211 221 231 311 321 331 0
-111 -121 0
-111 -131 0
-111 -211 0
-111 -221 0
-111 -231 0
-111 -311 0
-111 -321 0
-111 -331 0
-121 -131 0
-121 -211 0
-121 -221 0
```

```
-121 -231 0
-121 -311 0
-121 -321 0
-121 -331 0
-131 -211 0
-131 -221 0
-131 -231 0
-131 -311 0
-131 -321 0
-131 -331 0
-211 -221 0
-211 -231 0
-211 -311 0
-211 -321 0
-211 -331 0
-221 -231 0
-221 -311 0
-221 -321 0
-221 -331 0
-231 -311 0
-231 -321 0
-231 -331 0
-311 -321 0
-311 -331 0
-321 -331 0
```

We can see this is another list of 37 clauses. In fact this is the case for all fields in all blocks. Therefore we can say that the total number of clauses needed to express the Sudoku rules is 2997 * 4 = 11988.

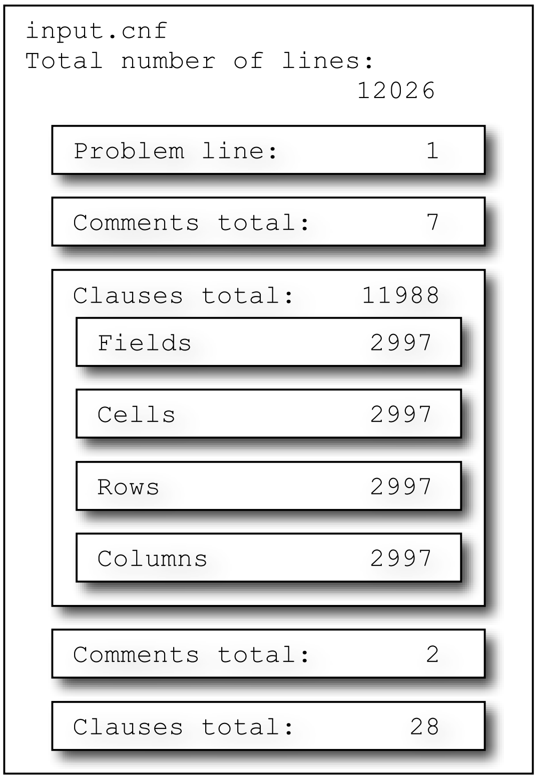


Fig.11. Input file with line numbers

Now we can define the problem line. If we take the example from above with 28 numbers already given in the puzzle then our line will look like this:

```
p cnf 729 12016
```

Where 12016 is the total number of clauses, so 11988 + 28 and 729 the total number of variables used.

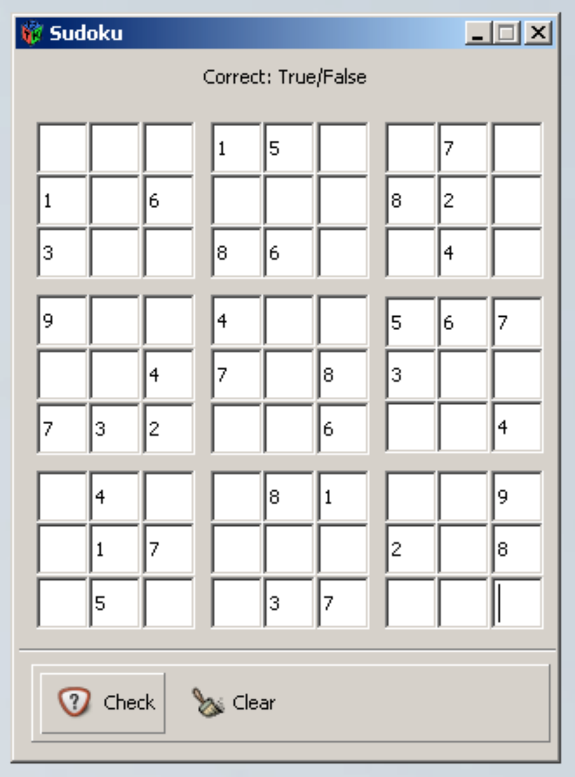
Since we write one clause per line, just so it is more readable, our example file will be 12026 lines long!! It is clear to see that the clauses describing the rules for the Sudoku puzzle are fixed, meaning they will stay the same for any instance of the game. This means that only the first line and the last part need to be generated each time. For my conversion program I have written those rules in a template.cnf file which I access and retrieve the data from each time I need to create a new input file for a new game.

There are actually different ways to describe the Sudoku rules. One of which I have put in a second template file so it can be used for testing or simply as an alternative. This second file only contains 3159 clauses but is not as straight forward and easily understandable so I have decided to make the other one the default.

I have taken both sets of clauses from the nice little tool "The Sudoku Puzzle as a Satisfiability Problem" written by Ivor Spence [17]. It allows users to see the actual encoding of a puzzle. So I have taken the freedom of copying his set of clauses instead of writing out 11996 lines of clauses by myself. It has spared me from trying to correct many many typos and errors in my writing. So, in that sense, thank you Ivor Spence for this lovely tool.

4.2 Summary of Prototype

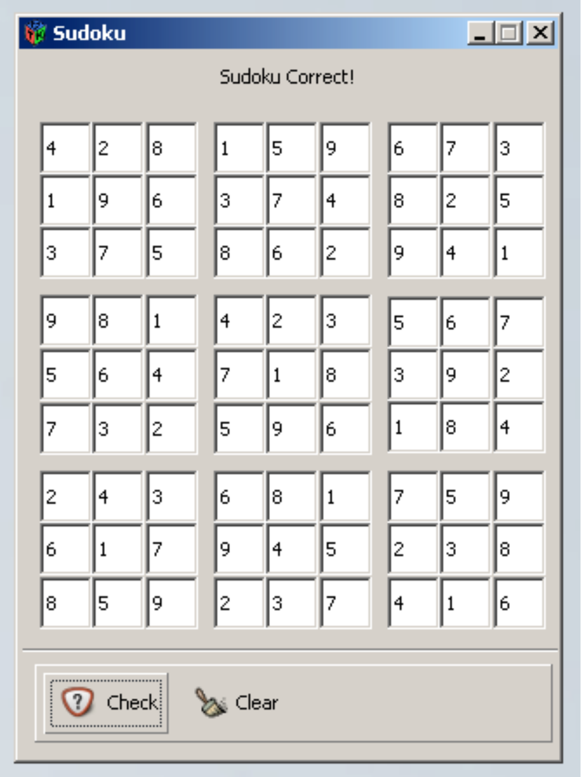
The prototype, submitted with the Interim Report at the end of the first semester, covered three of the main problems I was dealing with. These problems were building a user interface using Haskell, dealing with some I/O actions in Haskell and also the implementation of a simple "check" function which can verify if a filled Sudoku puzzle is correct or not. I will summarize the main points the prototype covered and discuss how it build a basis for my final software product.



Correct: True/False

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

Fig.12. GUI with initial Sudoku



Sudoku Correct!

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

Fig.13. GUI with check button pressed

I had a rough GUI for my Sudoku Game as you can see above in figures 12 and 13. The user could enter some numbers by keyboard in the 9x9 grid on the window. After solving the puzzle by hand the user could press the check button in the lower left corner of the interface and the program returned a short message to the user at the top saying whether the puzzle was correct or not (see figure 13). The user could then click on the clear button to delete all entries in the grid and the message at the top and start all over if he liked. If a Sudoku was incorrect then the program just returned a message saying that the puzzle was wrong, it did not tell the user where his mistakes were. Also, the user could not use the check button to check individual moves. If pressed before all fields in the grid were filled the program produced an error and closed down. These issues have been addressed in the final product and several changes and additions have been made.

4.3 Final Software Product

The general file architecture of my final software product can be explained as follows. The executable and the Haskell files are in the main directory together with the glade file and a readme information file. We also find a folder called MiniSat which contains the SAT Solver files. Inside that folder, next to all the MiniSat files we can find another folder called cnf and another readme file. The cnf folder contains the template for the cnf input file. The input.cnf and the output.cnf will be created and overwritten here.

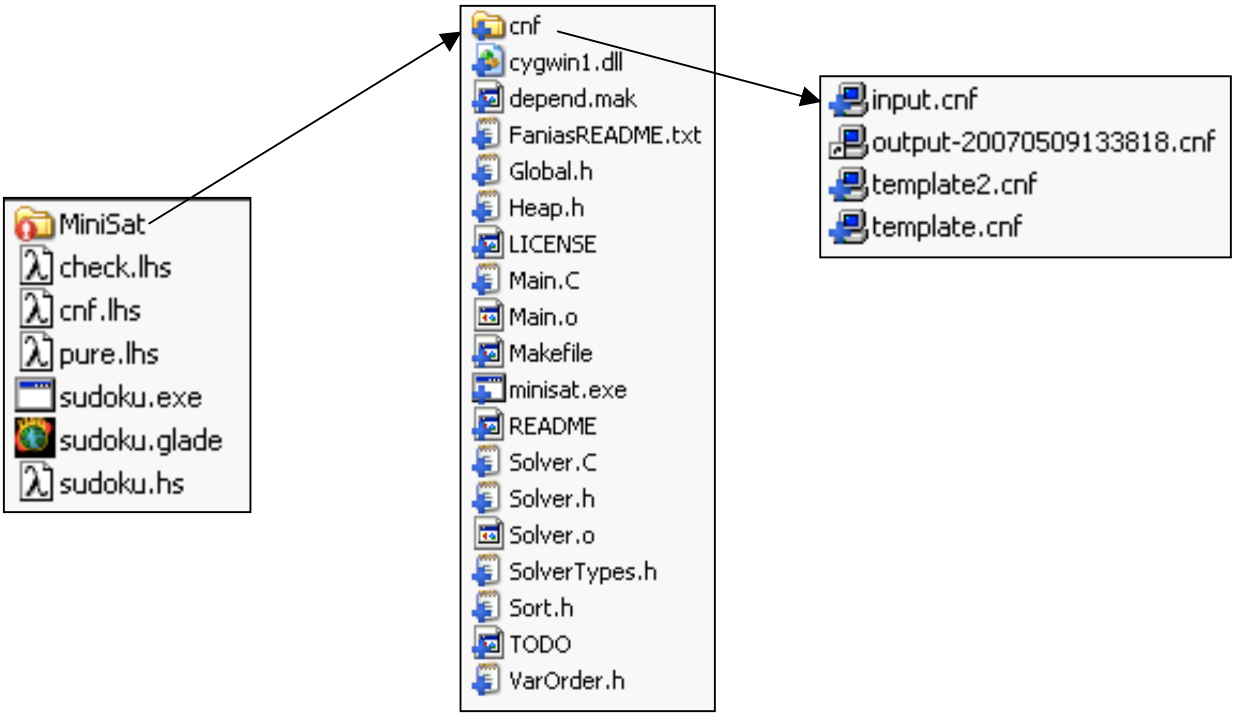


Fig.14. File Architecture of my Software

On the CD there will also be a general readme file with instructions on how to install and use my program.

Note that the output file for the SAT Solver has a unique timestamp in its filename, which means this file will not be overwritten and allows us to track the output of previous executions.

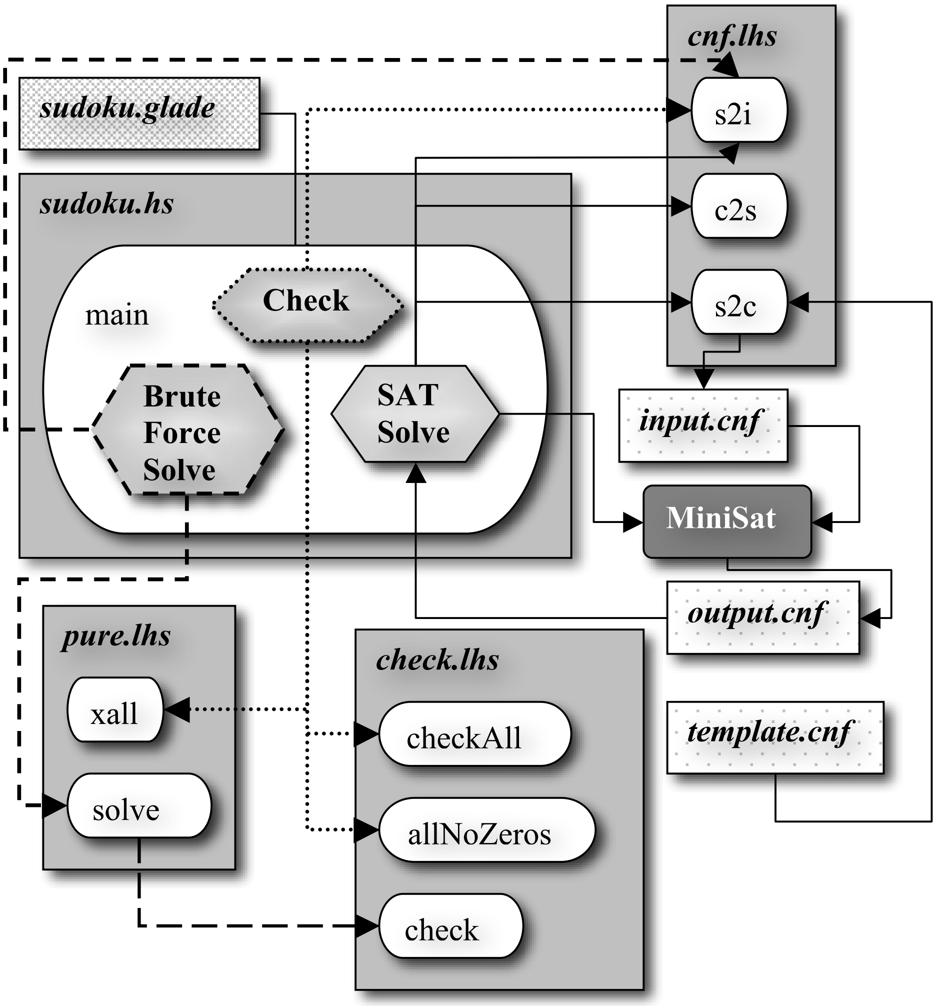
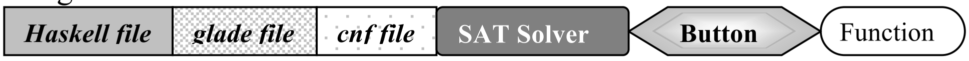


Fig.15. File Structure and Interaction

The way my files interact with each other is shown in figure 15. You can see the four Haskell files with the most important functions inside them. The "main" function inside the sudoku.hs file also shows three of its button actions. This is because those three buttons actually trigger a sequence of actions that include some interaction with other files. Each button has lines in a style unique to this specific button going out of it. These lines stand for some kind of interaction. Either it stands for using a function of another Haskell file, executing MiniSat or reading/accessing a file. It should be fairly simple to understand this graph with the legend given below.

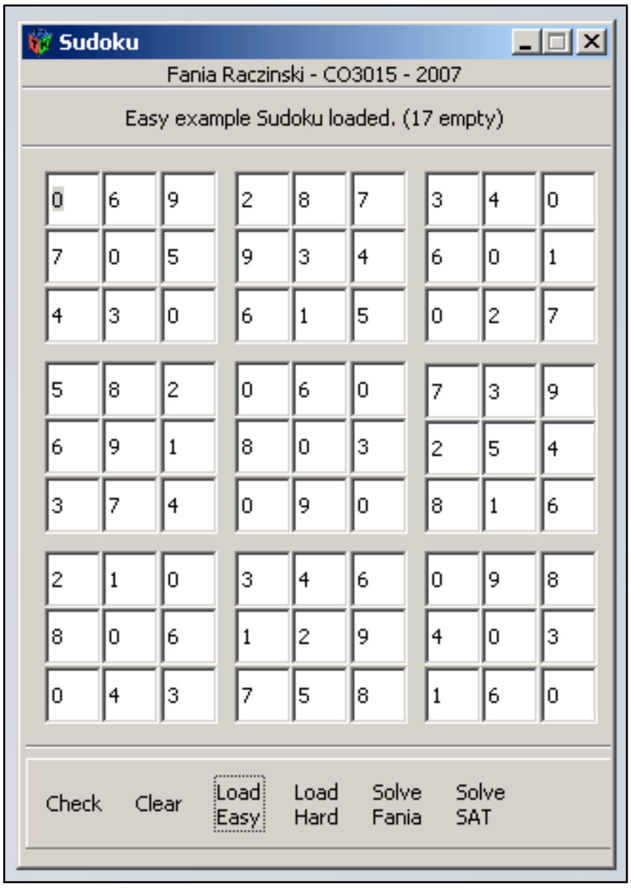
Legend



4.3.1 sudoku.hs

This is the main file of my software. It deals with all the user input, calls the relevant functions, creates and reads files, executes the SAT Solver when needed and so on. To explain this file and the structure of my program best I will divide it into its main functions or actions instead of going through my code sequentially. The few areas I will explain in detail are the design of the GUI, some principle code related to the interaction with Glade and the actions triggered by the few buttons available on the interface.

4.3.1.1 Graphical User Interface & Glade

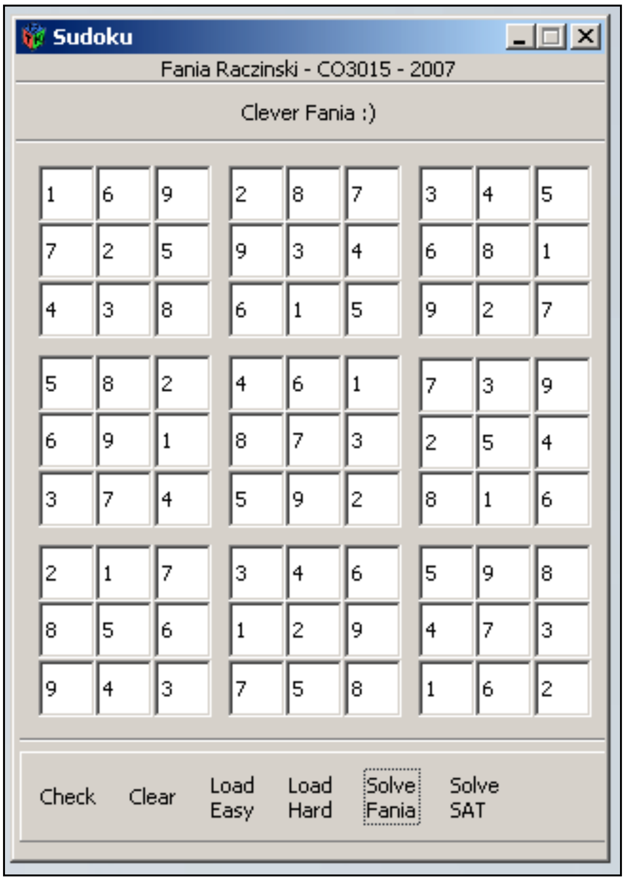


Fania Raczinski - CO3015 - 2007

Easy example Sudoku loaded. (17 empty)

0	6	9	2	8	7	3	4	0
7	0	5	9	3	4	6	0	1
4	3	0	6	1	5	0	2	7
5	8	2	0	6	0	7	3	9
6	9	1	8	0	3	2	5	4
3	7	4	0	9	0	8	1	6
2	1	0	3	4	6	0	9	8
8	0	6	1	2	9	4	0	3
0	4	3	7	5	8	1	6	0

Fig.16. Easy example puzzle loaded



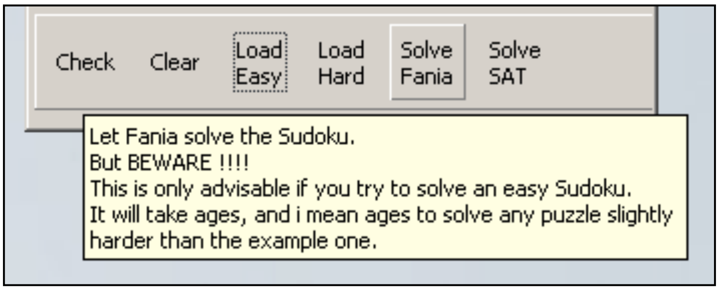
Fania Raczinski - CO3015 - 2007

Clever Fania :)

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

Fig.17. After pressing Solve Fania Button

The Interface has not changed much compared to what the prototype looked like. There are just a few more buttons and another label right on the top of the interface, displaying some general information. I have added two buttons to load an easy and a hard example puzzle and two buttons for the Brute Force Solver and the SAT Solver respectively. The feedback label returns appropriate messages depending on the button pressed. I have also added some tool tips to all of the buttons in order to clarify what they do.



Let Fania solve the Sudoku.
But BEWARE !!!!
This is only advisable if you try to solve an easy Sudoku.
It will take ages, and i mean ages to solve any puzzle slightly harder than the example one.

Fig.18 Tool Tipp for Solve Fania Button

The last thing I should mention is that due to some internal programming issues in each empty field there has to be a 0 instead of nothing. But I will explain this in detail later on.

To get a better idea of how this program works I have to explain Glade first. Glade lets a developer build user interfaces using a visual tool. It then produces a simple XML file in which all widgets (GUI components) are listed with their own unique ID’s and attributes. By using Gtk2Hs the developer can then access these widgets by calling their ID’s from the relevant XML file.

To get started we need to import the necessary libraries.

```
-- GUI
import Graphics.UI.Gtk
import Graphics.UI.Gtk.Glade
-- External commands
import System.Cmd
import System.Directory
-- For isDigit Check
import Data.Char
-- For timestamp
import Time
import Locale
-- Others
import Pure
import CNF
import Check
```

Then we need to initialise the interface and access the glade XML file of our GUI and assign it to a variable for the ease of further use. This is done in the “main” function of the Haskell module.

```
main = do
      -- Initialise GUI
      initGUI
      -- Get the glade xml file and do some error handling
      xmlM <- xmlNew "sudoku.glade"
      let xml = case xmlM of
                  (Just xml) -> xml
                  Nothing -> error "Cannot find .glade file in
                                    current directory"
```

The first thing we need to access in our glade XML file, now accessible using the variable "xml", is the window widget. We obviously want to tell Haskell what it has to display. "window1" is the ID of our window widget as given in our glade file. We let Haskell know that it is of widget type "window" by calling the function "castToWindow". The function "xmlGetWidget" literally does what it claims to do; it gets a widget from the input XML file, and in our case the input XML file is the one assigned to the xml variable. The only thing left to do is to assign this new widget we just extracted from our glade file to a variable we can use in the rest of our "main" function. This variable is "window".

```
-- Get the main window
window <- xmlGetWidget xml castToWindow "window1"
```

The next thing we can do is already some kind of I/O action. We tell the program that if the close button of the window widget is pressed it should exit the "main" function and quit the program. The function "onDestroy" takes a widget, in this case our window, and the action it is supposed to be doing if called.

```
-- Quit program if window is closed
onDestroy window mainQuit
```

To finish the "main" function we write the following. This line will actually be the very last line of the "main" function but because there are many things to be taken care of before that and for the purpose of completeness I have chosen to put this here.

```
mainGUI
```

The next step we do is to deal with lots of other things like assigning each remaining widget to a variable and all of the IO actions when certain buttons are pressed.

Let's assume the user fills in all the fields of the grid and presses the Check button. What is it we want the program to do? Of course we expect that it returns some kind of message notifying us whether the puzzle was correct or not. I already explained the general procedure of basic I/O action above when I explained how to deal with the user hitting the close button of the window. In this case though we somehow need to deal with actual input of the user, not just some button pressing. In fact we need to know where and what the user has entered into each of the 81 fields of the Sudoku grid. So we need to assign 81 variables to the corresponding text entry widgets in the glade XML file. In this case we obviously don't cast them to be of type Window but of type Entry instead.

```
-- Get all 81 entry fields of the Sudoku board
r1c1 <- xmlGetWidget xml castToEntry "entry1"
r1c2 <- xmlGetWidget xml castToEntry "entry2"
r1c3 <- xmlGetWidget xml castToEntry "entry3"
…
r6c4 <- xmlGetWidget xml castToEntry "entry49"
r6c5 <- xmlGetWidget xml castToEntry "entry50"
r6c6 <- xmlGetWidget xml castToEntry "entry51"
…
r9c7 <- xmlGetWidget xml castToEntry "entry79"
r9c8 <- xmlGetWidget xml castToEntry "entry80"
r9c9 <- xmlGetWidget xml castToEntry "entry81"
```

We also need to assign variables for the six buttons we use so we can actually call them at a later stage and tell them what to do if they are pressed.

```
-- Get the buttons below the board
checkbutton <- xmlGetWidget xml castToButton "button1"
clearbutton <- xmlGetWidget xml castToButton "button2"
loadbutton  <- xmlGetWidget xml castToButton "button3"
solvebutton <- xmlGetWidget xml castToButton "button4"
easybutton  <- xmlGetWidget xml castToButton "button6"
faniabutton <- xmlGetWidget xml castToButton "button5"
```

The same needs to be done to the text widget where we want to display our feedback messages to the user.

```
-- Get the info label above the board
feedback <- xmlGetWidget xml castToLabel "label1"
```

4.3.1.2 getEntries

This isn't an actual function or anything like it, but it's the name for a list of a series of actions I need to use repeatedly and I decided to treat with some special care. Basically it is used to get a list of all the current values in the grid at the time the check button is pressed or one of the two solve buttons.

```
let getEntries = [do { -- Get value of each entry field
                     ; oneone   <- entryGetText r1c1
                       ...
                     ; ninenine  <- entryGetText r9c9
                     ; -- Make a list for each row
                     ; let row1 = [oneone,onetwo,onethree,onefour,
                                   onefive,onesix,oneseven,
                                   oneeight,onenine]
                       ...
                     ; let row9 = [nineone,ninetwo,ninethree,
                                   ninefour,ninefive,ninesix,
                                   nineseven,nineeight,ninenine]
                     ; -- A list for all rows
                     ; let allRows = [row1,row2,row3,row4,row5,
                                      row6,row7,row8,row9]
                     ; return allRows
                     }
                  ]
```

Whenever we need the current values of the grid we can just simply execute these actions in a sequence (actually making use of the predefined function "sequence" of the Haskell prelude library [31]) and store the result in a new variable as follows.

```
-- Get the current values in the board
rows <- sequence getEntries
let allRows = head rows
```

4.3.1.3 Check Button

When the check button is pressed we want to get all current values of the grid and see whether they satisfy the Sudoku rules or not. In order to do this we need to use the getEntries procedure as explained above first. The next step we do is to get a list containing Boolean values for each of the 81 fields. Each Boolean is going to be set to true if the value is a digit and false otherwise. This list is used to check whether any letters instead of numbers have been entered and some error handling can be done.

```
onClicked checkbutton $ do
```

```
…
```

```
-- A list of Booleans. True if a digit and False if a char.
let checkrows = concat $ map (map isDigit) (map (map head) allRows)
```

Now we can actually do a check if this list only contains Booleans of value true or not. This is done using the predefined Haskell function “and” [31]. If this returns false, meaning not all of the values are true and hence the grid contains some other values than digits, then the program returns some error message in the feedback label. If not and only digits have been entered then we do several actions. First we create a new list where we convert each of the values into an int using the “s2i” function of the CNF module (I will explain this function in detail later on) and the predefined function “map” [31]. We then use this list to create a new list of all rows, all columns and all cells with the help of the “xall” function of the Pure module (again, I will explain this in the relevant chapter later). With this allInOne list we can actually perform the final check. We first get rid of all zeros in the list (using the “allNoZeros” function) in case the check button is used to check on an individual move and then we check if this list is valid according to the Sudoku rules or not. In both cases we return an appropriate message to the user in the feedback label above the grid.

```
-- Check that only numbers are entered. If not return error.
if (and checkRows) == False
      then do labelSetText feedback "Shmoo! Only enter numbers
                                     please..."
      else do {
            ; -- Make a list for all rows converting each
            ; -- element into an int.
            ; let allIntRows = map (map CNF.s2i) allRows
            ;
            ; -- Make a list for all rows, columns and cells.
            ; let allInOne = Pure.xall allIntRows
            ;
            ; -- Check if the current set of values is ok
            ; -- according to the Sudoku rules.
            ; if Check.checkAll $ Check.allNoZeros allInOne
                  then do labelSetText feedback "Correct!"
                  else do labelSetText feedback "WRONG..."
            }
```

4.3.1.4 Clear and Load Buttons

The clear button and the two load buttons do essentially the same thing so I will explain them in one go. We use the function “entrySetText” to set the value of each entry field in the grid to either “0” or the corresponding number of an example Sudoku. In addition we also set the message displayed in the feedback label to something appropriate. For the clear button this would look something like this.

```
onClicked clearbutton $ do
      entrySetText r1c1 "0"
      ...
      entrySetText r9c9 "0"
      labelSetText feedback ""
```

4.3.1.5 Brute Force Solver Button

```
onClicked faniabutton $ do
```

First what we need to do is to get the current values of the grid again using getEntries as described above. Also we do the same error handling as we did with the check button in case the user inputs a char instead of a number for example. If that check turns out to be ok then we can actually start the real action. The following code fragment is actually inside this part of the if statement (if only digits were entered). The actual important bit here is when we call the function “solve” of the Pure module. The list solved contains the final values of the finished Sudoku. Depending on the difficulty of the puzzle this can take a while. I will explain this function in the corresponding chapter though. We then check if this list is empty or not. If it is empty it means the Sudoku did not have a solution and we return an error message to the feedback label. If it is not empty then we simply set the values inside the text fields of the grid to the correct values of the solved puzzle and return a positive message to the user.

```
-- Make a list for all rows converting each
-- element into an int
let allIntRows = map (map CNF.s2i) allRows

let solved = concat $ concat $ Pure.solve allIntRows
print solved

-- Do some error handling in case sudoku is wrong
-- otherwise set board to new values
if (length solved) == 0
      then do {
              ; print "Baaah cant solve this..."
              ; labelSetText feedback "Unsolvable :("
              }
      else do {
              ; print "Yaayyy super Fania"
              ; labelSetText feedback "Clever Fania :)"
              ; entrySetText r1c1 (show $ head solved)
              ; entrySetText r1c2 (show $ head $ drop 1 solved)
              ; entrySetText r1c3 (show $ head $ drop 2 solved)
              ; entrySetText r1c4 (show $ head $ drop 3 solved)
                ...
              ; entrySetText r9c8 (show $ head $ drop 79 solved)
              ; entrySetText r9c9 (show $ head $ drop 80 solved)
              }
```

4.3.1.6 SAT Solver Button

```
onClicked solvebutton $ do
```

Same as before with the Brute Force Solver Button we get the current values of the grid, do some error handling to make sure only digits are entered and look in detail inside this if statement. We convert our list into a list of ints again and use this to create the input file for the SAT Solver. I will explain this in detail in the next chapter though. We then create a timestamp with the current date and time in it. Once we have the input file and this timestamp ready we save a string with the command to call the SAT Solver in a variable and change the current directory. Inside the MiniSat folder we can execute the command using the "system" function of the System.Cmd library. The SAT Solver will then execute and create the output file (with the timestamp in the filename) which we can read in and pass on to the CNF module to convert to a list of values – the solution to the puzzle given. Similar to above we handle with the case of the puzzle being unsolvable. We check the first word of the output file which is either "UNSAT" or "SAT". If it is unsatisfiable then we just return some error message and if it is satisfiable then we change the fields of the grid to the values of the solved puzzle and return a message of success to the feedback label. The last thing to do is only to change the directory back so we don't get an error next time we press this button.

```
-- Make a list for all rows converting each
-- element into an int
let allIntRows = map (map CNF.s2i) allRows

-- Create input.cnf file based on current list of values
CNF.s2c allIntRows

-- Get and print timestamp
t <- getClockTime
let time
  = formatCalendarTime defaultTimeLocale "%Y%m%d%H%M%S" (toUTCTime t)
let timestamp = "Timestamp: " ++ time
print timestamp

let cmd = "minisat cnf/input.cnf cnf/output-" ++ time ++ ".cnf"
setCurrentDirectory "MiniSat/"

-- Execute minisat command
system cmd

-- Read the output file
file <- readFile ("cnf/output-" ++ time ++ ".cnf")

-- Get the list of new values for the board
let solved = CNF.c2s file

-- Do some error handling in case Sudoku is wrong
-- otherwise set board to new values
if (head $ words file) == "UNSAT"
      then do {
              ; labelSetText feedback "MiniSat: Unsatisfiable :("
              }
      else do {
              ; labelSetText feedback "MiniSat: Satisfiable :)"
              ; entrySetText r1c1 (head solved)
              ; entrySetText r1c2 (head $ drop 1 solved)
              ; entrySetText r1c3 (head $ drop 2 solved)
```

```
                ...
              ; entrySetText r9c8 (head $ drop 79 solved)
              ; entrySetText r9c9 (head $ drop 80 solved)
              }
setCurrentDirectory ".."
```

4.3.2 cnf.lhs

Please remember that in any code fragments I am demonstrating here you will find the ">" character at the beginning of each line. This is a feature of literate Haskell files as I have explained in Chapter 4.1.1.

This module contains the code for converting to and from the DIMACS CNF format, producing the input.cnf file and reading the output.cnf file created by the SAT Solver. The first thing we need to do before anything else is give our module a suitable name and we import a library we will need later.

```
> module CNF where

> import Data.Char
```

4.3.2.1 Sudoku to CNF

In order to create the input file for the SAT Solver we need the list of current values of the grid. The following function takes this list and creates the input.cnf file. To do this it makes use of several other functions which I will explain roughly first and then later in detail.

```
> s2c xs = readFile "MiniSat/cnf/template.cnf" >>= \ s ->
>            writeFile "MiniSat/cnf/input.cnf" ((firstLine xs) ++
                                               "\n" ++ s ++ "\n" ++
                                              (rest xs))
```

To simplify this lets look at the following line.

```
readFile x >>= \ s -> writeFile input.cnf (firstLine ++ s ++ rest)
```

First we read the template file x (containing the clauses to describe the Sudoku rules) and pass it into the variable s using the >>= operator. The >>= operator, or monad to be specific, composes two actions sequentially, passing any value produced by the first as an argument to the second. The second action we do is to construct the input file and write it into input.cnf.

```
\ s -> writeFile input.cnf (firstLine ++ s ++ rest)
```

To understand this bit I need to explain the lambda notation. To the left of the arrow we can find arguments for a function and to the right we find the function definition or the result. This notation allows us to define functions within functions. The function "writeFile" takes two arguments; the first is the name of the file it should write to and the second is the value it is going to write. This second argument is composed of three strings. "firstline" is a function that creates the problem line for the file and "rest" creates the clauses for the current values in the grid. I will explain these two functions in detail after this. The variable s still contains the template data for the Sudoku rules from before. All together this forms the input file for the SAT Solver.

Now, let me explain how the “firstline” function works.

```
p cnf NumberOfVariables NumberOfClauses
```

The only thing we need, to construct this line, is the total number of clauses. This is the number of all clauses that describe the Sudoku rules (11988) plus the number of digits already given in the grid since each digit given will need one clause in this file. To calculate this number we make use of the “clauses” function which I will explain next.

```
> firstLine :: [[Int]] -> String
> firstLine xs =
               "p cnf 729 " ++ show (11988 + length (clauses xs 1 1))
```

This function takes the list of current values in the grid, an index for the row and an index for the column and returns the list of clauses. We go recursively through the given list and apply the function “r2s” (which I will explain below) to the head of the list and append the result to the list of strings. Since we have 9 rows in the given list we will go through this recursion 9 times in total, increasing the row index by one each time.

```
> clauses :: [[Int]] -> Int -> Int -> [String]
> clauses [] _ _ = []
> clauses (v:vs) x y = (r2s v x y) ++ (clauses vs (x+1) y)
```

The “r2s” function takes one row and the index for the current row and column and produces the clauses for each field in that row. If “v” is 0 then that field hasn’t been entered a value yet and can be skipped. “y” is increased by 1 each time a recursive call is made. If “v” has a value other than 0 then a string containing the current row, column and value is added recursively to the output list.

```
> r2s :: [Int] -> Int -> Int -> [String]
> r2s [] _ _ = []
> r2s (v:vs) x y
>     | v == 0    = r2s vs x (y+1)
>     | otherwise = (show x ++ show y ++ show v ++ " " ++ show 0):
                     r2s vs x (y+1)
```

4.3.2.2 CNF to Sudoku

The main function used to convert the output file the SAT Solver produces back to a list of strings containing the values for each field on the board is “c2s”. It uses several other functions so I will explain each one step by step. First it takes a string as input. This string is what has been read from the output.cnf file. 6 different functions are then performed on this string in sequence.

```
              6         5           4          3       2     1
> c2s x = map short $map i2s $filter (> 0) $map s2i $tail $words x
```

1: breaks the string up into a list of words, which were delimited by white space.
2: returns the list minus the first element (which is either "SAT" or "UNSAT") and leaves
   only the list of variables.

3: maps s2i to each element in the list, converting it to a list of ints instead of strings.
4: filter gets rid of all elements in the list which are negative or equal to 0.
5: maps i2s to each element in the list, turning it back into a list of strings.
6: maps short to each element in the list, which produces the final list.

Note that "$" has low right-associative binding precedence, allowing parentheses to be omitted.

The next three functions are very simple in themselves so ill only explain them briefly. "short" takes a string and drops the first two characters of it. "i2s" takes an int and converts it to a string and "s2i" does the opposite.

```
> short :: String -> String
> short x = drop 2 x

> i2s :: Int -> String
> i2s x = show x

> s2i :: String -> Int
> s2i x = read x :: Int
```

4.3.3 check.lhs

Please remember that in any code fragments I am demonstrating here you will find the > character at the beginning of each line. This is a feature of literate Haskell files as I have explained in Chapter 4.1.1.

Firstly, we need to name the module again.

```
> module Check where
```

Then we have three simple functions that allow us to check whether a puzzle or a move is correct or not. In order to be able to check individual moves we need to drop all 0s in the list we are working with. This is done using the "allNoZeros" function. It is a recursive function and uses a list comprehension to get rid of all 0s.

```
> allNoZeros :: [[Int]] -> [[Int]]
> allNoZeros []     = []
> allNoZeros (x:xs) = (noZeros x): allNoZeros xs
>                      where noZeros xs = [x | x<-xs, x/= 0]
```

The "check" function takes a list of integers and returns whether each member of this list only appears once. The "checkAll" function takes a list of lists (in our case it will take our allInOne list or rows, columns and cells) and checks though each element in the list recursively. I make use of the predefined function "elem" here [31]. It takes two inputs, an element and a list and checks whether that element is already a member of that list. If so, then it returns false. If not then it will return true. There is actually a predefined function that does that for us, so instead of using the code I have written I can simply use the function "all".

```
> check :: [Int] -> Bool
> check [x]    = True
> check (x:xs) = if elem x xs then False else check xs
```

```
checkAll :: [[Int]] -> Bool
checkAll [x]    = if check x then True else False
checkAll (x:xs) = if check x then checkAll xs else False

> checkAll xs = all check xs
```

4.3.4 pure.lhs

Please remember that in any code fragments I am demonstrating here you will find the > character at the beginning of each line. This is a feature of literate Haskell files as I have explained in Chapter 4.1.1.

This file contains my Brute Force Solver. It is called by the sudoku.hs file if the relevant button is pressed but it can also work as a stand alone terminal based solver. In this case we can just run "solve" with a list of rows, where each row is a list of ints. One thing I should mention is that this solver is very slow and depending on the difficulty of the puzzle can take unreasonably long to calculate the solution, but I will get back to this problem in another chapter.

First thing as usual is to name the module and import necessary libraries and modules. Also we declare a new type called row which is a list of ints.

```
> module Pure where

> import List
> import Check

> type Row = [Int]
```

The "solve" function takes a list of current rows and returns a list of solutions. This is achieved by using a list comprehension. The resulting list contains a list of rows [r1,r2,r3,r4,r5,r6,r7,r8,r9] with "r1" being an element of the list returned by the function "rows" applied to "i1" and so on. To make sure we only get valid Sudoku solutions we need to check this list. We do that with the function "checkAll" from the Check module. In addition we need to apply the function "xall" to the list first. It basically takes a list of rows and returns a list of rows, columns and cells to which we then can apply the "checkAll" function.

```
> solve :: [Row] -> [[Row]]
> solve [i1,i2,i3,i4,i5,i6,i7,i8,i9] =
>   [[r1,r2,r3,r4,r5,r6,r7,r8,r9] | r1 <- (rows i1), r2 <- (rows i2),
>                                   r3 <- (rows i3), r4 <- (rows i4),
>                                   r5 <- (rows i5), r6 <- (rows i6),
>                                   r7 <- (rows i7), r8 <- (rows i8),
>                                   r9 <- (rows i9),
>                  Check.checkAll $ xall [r1,r2,r3,r4,r5,r6,r7,r8,r9]]
```

Now to fully understand this we need to see how "rows" works. It takes one row and returns all possible solutions for that specific row. To do this we need to compare each element in the given list [i1,i2,i3,i4,i5,i6,i7,i8,i9] with each element inside the list comprehension [a,b,c,d,e,f,g,h,i]. If the element “a” is equal to 0 then it’s an empty field in the grid and can be any of the numbers from 1 to 9. If it is not 0 then it needs to stay the same number it was. In addition we check that each [a,b,c,d,e,f,g,h,i] is true according to the “check” function.

```
> rows :: Row -> [Row]
> rows [i1,i2,i3,i4,i5,i6,i7,i8,i9] =
>   [[a,b,c,d,e,f,g,h,i] | a <- (if (i1 == 0) then [1..9] else [i1]),
>                          b <- (if (i2 == 0) then [1..9] else [i2]),
>                          c <- (if (i3 == 0) then [1..9] else [i3]),
>                          d <- (if (i4 == 0) then [1..9] else [i4]),
>                          e <- (if (i5 == 0) then [1..9] else [i5]),
>                          f <- (if (i6 == 0) then [1..9] else [i6]),
>                          g <- (if (i7 == 0) then [1..9] else [i7]),
>                          h <- (if (i8 == 0) then [1..9] else [i8]),
>                          i <- (if (i9 == 0) then [1..9] else [i9]),
>                          Check.check [a,b,c,d,e,f,g,h,i]]
```

The “xall” function I have mentioned above is fairly easy.

```
> xall :: [Row] -> [Row]
> xall x = concat [x, transpose x, cells x]
```

You will notice it uses the function “cells” and I will explain this now. “cells” takes a list of rows and returns a list of cells. But this is much easier to understand if we look at the following figures first. This show how the cells are ordered and how I was able to get the values for each cell. Each cell is made up of three blocks with sets of three values.

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

Fig.19. Order of cells

Values 1,2,3	Values1,2,3	Values 1,2,3
Values 4,5,6	Values 4,5,6	Values 4,5,6
Values 7,8,9	Values 7,8,9	Values 7,8,9
Values 1,2,3	Values 1,2,3	Values 1,2,3
Values 4,5,6	Values 4,5,6	Values 4,5,6
Values 7,8,9	Values 7,8,9	Values 7,8,9
Values 1,2,3	Values 1,2,3	Values 1,2,3
Values 4,5,6	Values 4,5,6	Values 4,5,6
Values 7,8,9	Values 7,8,9	Values 7,8,9

Fig.20. Order of individual values in each cell

take 3 $ head x	take 3 $ drop 3 $ head x	drop 6 $ head x
take 3 $ head $ drop 1 x	take 3 $ drop 3 $ head $ drop 1 x	drop 6 $ head $ drop 1 x
take 3 $ head $ drop 2 x	take 3 $ drop 3 $ head $ drop 2 x	drop 6 $ head $ drop 2 x
take 3 $ head $ drop 3 x	take 3 $ drop 3 $ head $ drop 3 x	drop 6 $ head $ drop 3 x
take 3 $ head $ drop 4 x	take 3 $ drop 3 $ head $ drop 4 x	drop 6 $ head $ drop 4 x
take 3 $ head $ drop 5 x	take 3 $ drop 3 $ head $ drop 5 x	drop 6 $ head $ drop 5 x
take 3 $ head $ drop 5 x	take 3 $ drop 3 $ head $ drop 5 x	drop 6 $ head $ drop 5 x
take 3 $ head $ drop 6 x	take 3 $ drop 3 $ head $ drop 6 x	drop 6 $ head $ drop 6 x
take 3 $ head $ drop 7 x	take 3 $ drop 3 $ head $ drop 7 x	drop 6 $ head $ drop 7 x

Fig.21. How to get the values from the list of rows x

The code for this function is as follows.

```
> cells :: [Row] -> [Row]
> cells [] = []
> cells x = [ concat [(take 3 $ head x),
>                     (take 3 $ head $ drop 1 x),
>                     (take 3 $ head $ drop 2 x)],
>             concat [(take 3 $ drop 3 $ head x),
>                     (take 3 $ drop 3 $ head $ drop 1 x),
>                     (take 3 $ drop 3 $ head $ drop 2 x)],
>             concat [(drop 6 $ head x),
>                     (drop 6 $ head $ drop 1 x),
>                     (drop 6 $ head $ drop 2 x)],
>             concat [(take 3 $ head $ drop 3 x),
>                     (take 3 $ head $ drop 4 x),
>                     (take 3 $ head $ drop 5 x)],
>             concat [(take 3 $ drop 3 $ head $ drop 3 x),
>                     (take 3 $ drop 3 $ head $ drop 4 x),
>                     (take 3 $ drop 3 $ head $ drop 5 x)],
>             concat [(drop 6 $ head $ drop 3 x),
>                     (drop 6 $ head $ drop 4 x),
>                     (drop 6 $ head $ drop 5 x)],
>             concat [(take 3 $ head $ drop 6 x),
>                     (take 3 $ head $ drop 7 x),
>                     (take 3 $ head $ drop 8 x)],
>             concat [(take 3 $ drop 3 $ head $ drop 6 x),
>                     (take 3 $ drop 3 $ head $ drop 7 x),
>                     (take 3 $ drop 3 $ head $ drop 8 x)],
>             concat [(drop 6 $ head $ drop 6 x),
>                     (drop 6 $ head $ drop 7 x),
>                     (drop 6 $ head $ drop 8 x)]
>           ]
```

4.3.5 Example input.cnf

We can say this file is made up of 3 parts and the middle part is divided into 4 blocks in total.

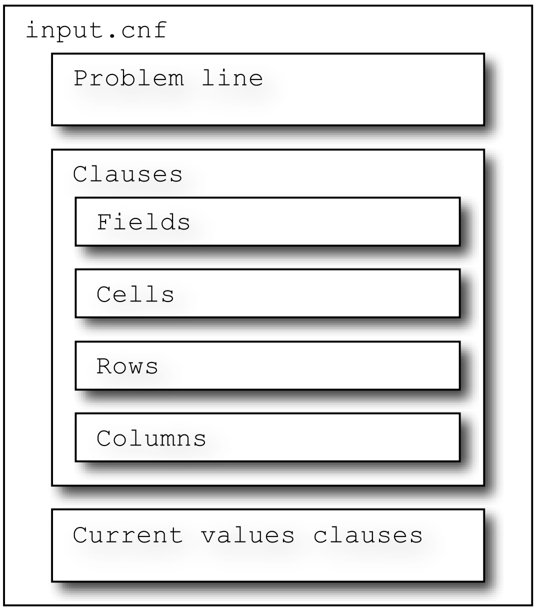


Fig.22. Design of input.cnf file

This is the first part of the file. It contains the problem line and some comments.

```
p cnf 729 12028
c Start of template.
c --------------------------------------------------
c The following lines describe the Sudoku rules.
c Each field can have one of the values of 1 to 9.
c The numbers of 1 to 9 must appear in each row,
c column and cell but only once.
c This template contains 11988 clauses.
```

This is followed by the 4 blocks of the middle part. The first block contains the clauses for each of the 81 fields. This bit of code is to describe the first field in row 1 and column 1. It needs to be repeated and obviously changed for the remaining 80 fields. This means the whole block to describe the rules for each field is 81 fields * 37 clauses, which is 2997 clauses long.

```
111 112 113 114 115 116 117 118 119 0
-111 -112 0
-111 -113 0
-111 -114 0
-111 -115 0
-111 -116 0
-111 -117 0
-111 -118 0
-111 -119 0
-112 -113 0
-112 -114 0
-112 -115 0
-112 -116 0
-112 -117 0
-112 -118 0
-112 -119 0
-113 -114 0
-113 -115 0
-113 -116 0
-113 -117 0
-113 -118 0
-113 -119 0
-114 -115 0
-114 -116 0
-114 -117 0
-114 -118 0
-114 -119 0
-115 -116 0
-115 -117 0
-115 -118 0
-115 -119 0
-116 -117 0
-116 -118 0
-116 -119 0
-117 -118 0
-117 -119 0
-118 -119 0
```

We need to do the same to express the rules for each cell, each row and each column too. So, in total this is 2997 clauses for each of these blocks summing up to 11988 clauses. The last bit missing in the file are the clauses for the current values of the grid. The Sudoku puzzle it currently represents is shown in figure 23.

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

Fig.23. Current puzzle

```
c --------------------------------------------------
c End of template.
126 0
142 0
167 0
184 0
217 0
235 0
253 0
276 0
291 0
323 0
346 0
365 0
382 0
415 0
432 0
456 0
477 0
499 0
529 0
548 0
563 0
585 0
613 0
634 0
659 0
678 0
696 0
721 0
743 0
766 0
789 0
818 0
836 0
852 0
874 0
893 0
924 0
947 0
968 0
986 0
```

4.4 Software testing

I have performed several tests on my program and its individual files. I will start off with explaining some tests on the actual program and then continue with a discussion of each file and its functions. The tests performed on individual functions have been done with GHCi.

I am using the following lists to test my functions.

example0Wrong	A list of invalid rows with some 0s in it.
exampleWrong	A list of invalid completed rows.
example0Right	A list of valid rows with some 0s in it.
exampleRight	A list of valid completed rows.
w0all	A list of invalid rows with 0s and the corresponding columns and cells.
wall	A list of invalid completed rows and the corresponding columns and cells.
r0all	A list of valid rows with 0s and the corresponding columns and cells.
rall	A list of valid completed rows and the corresponding columns and cells.

One of the things I should discuss beforehand is the 0 problem. When starting the program all fields in the grid are filled with a 0. This has two reasons. First, if those fields are left empty and we want to perform a check or solve the puzzle, such that the program will trigger the getEntries action discussed in chapter 4.3.1.2, the program will crash since it can't pass any value into a list. I could have worked around this by adding some if statements, saying that if a field has no value inside it then add a 0 to the list but I decided to leave this for the future development of the program. Several of my functions work with a list of all values currently in the grid. This means that I need to have some sort of placeholder value in case a field is empty. This is my 0. I should have added some sort of error catching in case a value is deleted by mistake and no 0 is written back into the field, but since it is closely connected to the problem above I also decided to leave this for the future development.

4.4.1 General Tests

Test	Expected result	Actual result	Success??
Close Button pressed.	Program terminates.	Program terminates.	Yes
Minimise Button pressed.	Program minimises.	Program minimises.	Yes
Some characters entered instead of numbers and check button or any of the two solve buttons pressed.	Error message displayed in feedback area.	Error message displayed in feedback area.	Yes
Field left empty, no 0 inside and check button or any of the two solve buttons pressed.	Error message displayed in feedback area.	Program shuts down.	NO
Clear Button pressed.	Any values already inside the grid are replaced by a 0.	Any values already inside the grid are replaced by a 0.	Yes
One of the Load Buttons pressed.	An example Sudoku is loaded into the grid, any values already inside are replaced.	An example Sudoku is loaded into the grid, any values already inside are replaced.	Yes
The **easy** Sudoku puzzle loaded and Brute Force Button pressed.	Solution displayed in the grid and message shown in feedback box.	Solution displayed in the grid and message shown in feedback box.	Yes
The **hard** Sudoku puzzle loaded and Brute Force Button pressed.	Solution displayed in the grid and message shown in feedback box.	Program freezes. It is actually calculating the solution but this takes unknown amount of time.	NO
The **easy** Sudoku puzzle loaded and SAT Button pressed.	Solution displayed in the grid and message shown in feedback box.	Solution displayed in the grid and message shown in feedback box.	Yes
The **hard** Sudoku puzzle loaded and SAT Button pressed.	Solution displayed in the grid and message shown in feedback box.	Solution displayed in the grid and message shown in feedback box.	Yes
A random Sudoku puzzle entered by hand and Brute Force Button pressed.	Solution displayed in the grid and message shown in feedback box.	Depends on the difficulty of the puzzle. If it is very easy it will return the solution, if not then it will freeze for an unknown amount of time.	NO
A random Sudoku puzzle entered by hand and SAT Button pressed.	Solution displayed in the grid and message shown in feedback box.	Solution displayed in the grid and message shown in feedback box.	Yes
Pressing the Check Button while the grid is in an invalid state.	Error message displayed in feedback area and no changes made to the grid.	Error message displayed in feedback area and no changes made to the grid.	Yes
Pressing the Check Button while the grid is in a valid state.	Positive message displayed in the feedback box.	Positive message displayed in the feedback box.	Yes

4.4.2 Check Tests

4.4.2.1 allNoZeros

These test cases are copied directly from GHCi.

```
*Check> allNoZeros example0Wrong
[[1,6,9,2],[1,9,3,8,1],[8,5,7],[8,2,6,1,7,3],[6,9,5,4],[7,4,5,9,8,1],
[2,3,5],[8,5,2,9,7],[8,1,6,2]]
```

```
*Check> allNoZeros exampleRight
[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,4
,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9,
8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]
```

As expected the function deleted the 0s from the list containing 0s and didn't change anything on the other list.

4.4.2.2 check

```
*Check> check ro1
False
```

```
*Check> check wr1
False
```

```
*Check> check sr1
True
```

The check with "ro1" correctly returns False since the row contains 0s. Similarly the check on "wr1" returns False because the row contains the number 1 twice. The check on "sr1" returns True since this row contains each value only once.

4.4.2.3 checkAll

```
*Check> checkAll w0all
False
```

```
*Check> checkAll wall
False
```

```
*Check> checkAll r0all
False
```

```
*Check> checkAll rall
True
```

As expected the “checkAll” function which is supposed to check whether a completed Sudoku is valid or not returns False for all test cases apart for the last which is our correct list of rows, columns and cells.

4.4.3 CNF Tests

Many of the functions in this module are tested already in chapter 4.4.1 with the general tests of the main program, so I won’t do it again here. This concerns mainly the functions related to the creation of the input file for the SAT Solver. Some of the sub functions I will discuss here.

4.4.3.1 r2s

```
*CNF> r2s rr1 1 1
["111 0","126 0","139 0","142 0","158 0","167 0","173 0","184 0","195
0"]
```

```
*CNF> r2s rr2 2 1
["217 0","222 0","235 0","249 0","253 0","264 0","276 0","288 0","291
0"]
```

This function is used recursively inside the “clauses” function and it is quite interesting to see some test cases for it. The two examples above show how to use this function in order to get the DIMACS CNF clauses for row 1 and 2 of the valid and filled example rows “rr1” and “rr2”.

The last argument for the function is the column index, so just as an example let’s try and change that.

```
*CNF> r2s rr1 1 5
["151  0","166  0","179  0","182  0","198  0","1107  0","1113  0","1124
0","1135 0"]
```

The result is completely wrong. This is the reason we call this function with a 1 as the last argument each time.

Another test case is this:

```
*CNF> r2s ro1 1 1
["126 0","139 0","142 0","158 0","167 0","173 0","184 0"]
```

Here, we use an example row with some 0s in it and we can see that the result is as expected only containing the values for those fields not containing a 0.

Again another test is to use an invalid row, but this also yields to the expected result.

```
*CNF> r2s wo1 1 1
["111 0","126 0","139 0","142 0"]
```

4.4.3.2 short & i2s & s2i

```
*CNF> short "test"
"st"
```

```
*CNF> short "12345"
"345"
```

```
*CNF> i2s 1
"1"
```

```
*CNF> s2i "1"
1
```

All of these test cases return the expected result.

4.4.2 Pure Tests (Brute Force Solver)

To test the functions in this file I have made use of the profiling system of the Glasgow Haskell Compiler [29]. To profile a program we need to follow a three-step process:

1. Re-compile the program with the `-prof` option.
2. Run the program with one of the profiling options.
3. Examine the generated profiling information.

In order to compile the program into an executable we need a “main” function. We simply add this line and compile it. I will copy the relevant information from the pure.exe.prof file in here and discuss the results.

Compile and run for profiling:

```
$ ghc -prof -auto-all -o pure pure.lhs
$ ./pure +RTS –p
```

An example profiling file is shown below. The important parts are highlighted. And in the test cases that I will consider in this chapter I will only copy these relevant parts and explain them.

CAF stands for Constant Applicative Form and can be explained as follows [29]. Because Haskell is a lazy programming language certain expressions are only evaluated once, if at all. For example the expression `x = “Hello”++“ World!”` is only evaluated once and any subsequent calls for `x` will immediately get to see the cached result. The definition `x` is called a CAF because it has no arguments.

```
       Sun May 06 23:55 2007 Time and Allocation Profiling Report  (Final)

          pure.exe +RTS -p -RTS

       total time  =        0.00 secs   (0 ticks @ 50 ms)
       total alloc =       7,624 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE               %time %alloc

CAF                            GHC.Handle             0.0   12.5
rows                            Main                   0.0   55.7
ro5                            Main                   0.0    1.4
CAF                            Main                   0.0   30.3

                                                        individual    inherited
COST CENTRE   MODULE       no.    entries  %time %alloc   %time %alloc

MAIN          MAIN           1          0    0.0    0.0     0.0  100.0
 CAF          Main         156         18    0.0   30.3     0.0   87.5
  ro5         Main         164          1    0.0    1.4     0.0    1.4
  main        Main         162          1    0.0    0.1     0.0   55.8
   rows        Main         163          1    0.0    55.7     0.0   55.7
    check     Main         165         39    0.0    0.0     0.0    0.0
 CAF          GHC.Handle    91          4    0.0   12.5     0.0   12.5
```

4.4.2.1 rows

This function returns the list of all possible solutions for one single row. Let's start with an easy example.

One empty field:

```
> main = print $ rows ro5
> ro5 = [6,9,1,8,0,3,2,5,4]
```

```
$ ./pure +RTS -p
[[6,9,1,8,7,3,2,5,4]]
```

```
*Main> length $ rows ro5
1
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =       7,624 bytes  (excludes profiling overheads)
```

This is the easiest example since it only has one empty field that needs to be filled. It doesn't even take 0.00 seconds to solve this. But let us look at some more examples, increasing the number of empty fields by one each time.

Two empty fields:

```
> main = print $ rows ro1
> ro1 = [0,6,9,2,8,7,3,4,0]
```

```
$ ./pure +RTS -p
[[1,6,9,2,8,7,3,4,5],[5,6,9,2,8,7,3,4,1]]
```

```
*Main> length $ rows ro1
2
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =      21,576 bytes  (excludes profiling overheads)
```

Three empty fields:

```
> main = print $ rows wo6
> wo6 = [0,7,4,5,9,0,8,1,0]
```

```
$ ./pure +RTS -p
[[2,7,4,5,9,3,8,1,6],[2,7,4,5,9,6,8,1,3],[3,7,4,5,9,2,8,1,6],[3,7,4,5
,9,6,8,1,2],[6,7,4,5,9,2,8,1,3],[6,7,4,5,9,3,8,1,2]]
```

```
*Main> length $ rows wo6
6
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =     138,340 bytes  (excludes profiling overheads)
```

Four empty fields:

```
> main = print $ rows wo8
> wo8 = [8,5,0,0,2,9,0,7,0]
```

```
$ ./pure +RTS -p
[[8,5,1,3,2,9,4,7,6],[8,5,1,3,2,9,6,7,4],[8,5,1,4,2,9,3,7,6],[8,5,1,4
,2,9,6,7,3],[8,5,1,6,2,9,3,7,4],[8,5,1,6,2,9,4,7,3],[8,5,3,1,2,9,4,7,
6],[8,5,3,1,2,9,6,7,4],[8,5,3,4,2,9,1,7,6],[8,5,3,4,2,9,6,7,1],[8,5,3
,6,2,9,1,7,4],[8,5,3,6,2,9,4,7,1],[8,5,4,1,2,9,3,7,6],[8,5,4,1,2,9,6,
7,3],[8,5,4,3,2,9,1,7,6],[8,5,4,3,2,9,6,7,1],[8,5,4,6,2,9,1,7,3],[8,5
,4,6,2,9,3,7,1],[8,5,6,1,2,9,3,7,4],[8,5,6,1,2,9,4,7,3],[8,5,6,3,2,9,
1,7,4],[8,5,6,3,2,9,4,7,1],[8,5,6,4,2,9,1,7,3],[8,5,6,4,2,9,3,7,1]]
```

```
*Main> length $ rows wo8
24
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =   1,139,288 bytes  (excludes profiling overheads)
```

Five empty fields:

```
> main = print $ rows wo5
> wo5 = [6,9,0,0,0,0,0,5,4]
```

```
*Main> length $ rows wo5
120
```

```
total time  =        0.10 secs   (2 ticks @ 50 ms)
total alloc =  17,639,652 bytes  (excludes profiling overheads)
```

Six empty fields:

```
> main = print $ rows wo3
> wo3 = [0,0,8,0,0,5,0,0,7]
```

```
*Main> length $ rows wo3
720
```

```
total time  =        0.55 secs   (11 ticks @ 50 ms)
total alloc = 122,751,728 bytes  (excludes profiling overheads)
```

Seven empty fields:

```
> main = print $ rows xx7
> xx7 = [2,3,0,0,0,0,0,0,0]
```

```
*Main> length $ rows xx7
5040
```

```
total time  =        4.30 secs   (86 ticks @ 50 ms)
total alloc = 774,243,612 bytes  (excludes profiling overheads)
```

Eight empty fields:

```
> main = print $ rows xx8
> xx8 = [8,0,0,0,0,0,0,0,0]
```

```
*Main> length $ rows xx8
40320
```

```
total time  =       40.00 secs   (800 ticks @ 50 ms)
total alloc = 6,963,990,168 bytes  (excludes profiling overheads)
```

Nine empty fields:

```
> main = print $ rows xx9
> xx9 = [0,0,0,0,0,0,0,0,0]
```

```
*Main> length $ rows xx9
362880
```

```
total time  =      272.25 secs   (5445 ticks @ 50 ms)
total alloc = 62,675,891,548 bytes  (excludes profiling overheads)
```

These examples show very clearly how quick the time and space needed increases with each empty field more in a row. To calculate all 9! possibilities for one empty row alone it takes roughly **4.5 minutes** = 272.25 seconds!

4.4.2.2 solve

The time and space needed to calculate a full 9x9 Sudoku puzzle depends very much on how many empty fields there are and where these fields are located. I have chosen some very simple patterns for the following examples, with the empty fields in the first example being located as far from each other as possible and very close together in the second. Let me explain with some pictures.

0	6	9	2	**0**	7	3	4	**0**
7	**0**	5	9	3	4	6	**0**	1
4	3	**0**	6	1	5	**0**	2	7
5	8	2	**0**	6	**0**	7	3	9
0	9	1	8	**0**	3	2	5	**0**
3	7	4	**0**	9	**0**	8	1	6
2	1	**0**	3	4	6	**0**	9	8
8	**0**	6	1	2	9	4	**0**	3
0	4	3	7	**0**	8	1	6	**0**

Fig.24. example21empty

1	6	9	2	8	7	3	4	5
7	2	5	9	3	4	6	8	1
4	3	8	6	1	5	9	2	7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	9
0	**0**	**0**	**0**	**0**	**0**	**0**	5	4
0	**0**	**0**	**0**	**0**	**0**	**0**	1	6
2	1	7	3	4	6	5	9	8
8	5	6	1	2	9	4	7	3
9	4	3	7	5	8	1	6	2

Fig.25. bad21example

Figure 24 shows example21empty. This took about 2 seconds to be solved. If we would try to solve the example on the right though it would probably take us much longer, even though both examples have the same number of empty fields. This shows how much the position of them matters for the solving time.

A typical standard Sudoku puzzle of normal difficulty has about 46 empty fields. This is more than twice the amount. Imagine the minimal Sudoku with only 17 givens, which is equal to 64 empty fields!

So with the examples shown here it is safe to say it would take an unknown amount of time to solve any puzzle slightly harder then 17 empty fields and is not advisable to try.

Things that have an impact on the time and space needed to solve any puzzle:

- The number of empty fields
- The position of those empty fields

example21empty:

```
> main = print $ solve example21empty
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
4,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]]
```

```
total time  =        1.95 secs   (39 ticks @ 50 ms)
total alloc = 382,162,880 bytes  (excludes profiling overheads)
```

bad21example:

```
> main = print $ solve bad21example
```

```
$ ./pure +RTS -p
pure.exe: interrupted
```

Interrupted after 8.23 hours!

```
       Mon May 07 14:15 2007 Time and Allocation Profiling Report  (Final)

          pure.exe +RTS -p -RTS

       total time  =    29615.70 secs   (592314 ticks @ 50 ms)
       total alloc = 10272,513,537,988 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE               %time %alloc

check                          Main                  49.9    0.0
rows                            Main                  49.9   99.7

                                                  individual    inherited
COST CENTRE       MODULE          no.    entries  %time %alloc   %time %alloc

MAIN              MAIN              1          0    0.0    0.0   100.0  100.0
 CAF              Main            156          8    0.0    0.0   100.0  100.0
  bad21example    Main            164          1    0.0    0.0     0.0    0.0
  main            Main            162          1    0.0    0.0   100.0  100.0
   solve          Main            163          1    0.1    0.1   100.0  100.0
    xall          Main            168   35990640    0.1    0.2     0.1    0.2
    checkAll      Main            167   35990640    0.0    0.0     2.3    0.0
     check        Main            169 2951232480    2.2    0.0     2.2    0.0
    rows           Main            165   107979068   49.9   99.7    97.6
99.7
     check        Main            166 56444427819   47.7    0.0    47.7    0.0
 CAF              GHC.Handle       91          2    0.0    0.0     0.0    0.0
```

8.23 hours!!!

example0Wrong:

```
> main = print $ solve example0Wrong
```

```
$ ./pure +RTS -p
[]
```

```
total time  =        0.95 secs   (19 ticks @ 50 ms)
total alloc = 244,755,328 bytes  (excludes profiling overheads)
```

exampleWrong:

```
> main = print $ solve exampleWrong
```

```
$ ./pure +RTS -p
[]
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =       3,656 bytes  (excludes profiling overheads)
```

exampleRight:

```
> main = print $ solve exampleRight
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
4,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]]
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =      31,496 bytes  (excludes profiling overheads)
```

1 empty field:

```
> main = print $ solve example1empty
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
4,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]]
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =      35,052 bytes  (excludes profiling overheads)
```

9 empty fields:

```
> main = print $ solve example9empty
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
4,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]]
```

```
total time  =        0.00 secs   (0 ticks @ 50 ms)
total alloc =      60,044 bytes  (excludes profiling overheads)
```

17 empty fields:

```
> main = print $ solve example17empty
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
4,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2]]]
```

```
total time  =        0.05 secs   (1 ticks @ 50 ms)
total alloc =   5,710,500 bytes  (excludes profiling overheads)
```

25 empty fields:

```
> main = print $ solve example25empty
```

```
$ ./pure +RTS -p
[[[1,6,9,2,8,7,3,4,5],[7,2,5,4,3,9,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,
1,6,4,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9
,8],[8,5,6,9,2,1,4,7,3],[9,4,3,7,5,8,1,6,2]],[[1,6,9,2,8,7,3,4,5],[7,
2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,4,6,1,7,3,9],[6,9,1,8,7,3
,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9,8],[8,5,6,1,2,9,4,7,3],[
9,4,3,7,5,8,1,6,2]]]
```

```
       Mon May 07 02:10 2007 Time and Allocation Profiling Report  (Final)

          pure.exe +RTS -p -RTS

       total time  =      353.50 secs   (7070 ticks @ 50 ms)
       total alloc = 68,770,419,848 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE               %time %alloc

check                          Main                  70.4    0.0
rows                            Main                  28.7   97.8
xall                           Main                   0.7    1.7

                                                  individual    inherited
COST CENTRE        MODULE       no.    entries  %time %alloc   %time %alloc

MAIN               MAIN           1          0   0.0    0.0   100.0  100.0
 CAF               Main         156         10   0.0    0.0   100.0  100.0
  example25empty   Main         164          1   0.0    0.0     0.0    0.0
  main             Main         162          1   0.0    0.0   100.0  100.0
   solve           Main         163          1   0.1    0.5   100.0  100.0
    xall           Main         168    1990656   0.7    1.7     0.7    1.7
     cells         Main         170          2   0.0    0.0     0.0    0.0
    checkAll       Main         167    1990656   0.1    0.0    13.9    0.0
     check         Main         169  165404162  13.8    0.0    13.8    0.0
    rows            Main         165     356983  28.7   97.8    85.3   97.8
     check         Main         166  625422933  56.6    0.0    56.6    0.0
 CAF               GHC.Handle    91          4   0.0    0.0     0.0    0.0
```

5.9 minutes

4.4.2.3 xall & cells

```
*Main> xall exampleRight
[[1,6,9,2,8,7,3,4,5],[7,2,5,9,3,4,6,8,1],[4,3,8,6,1,5,9,2,7],[5,8,2,4
,6,1,7,3,9],[6,9,1,8,7,3,2,5,4],[3,7,4,5,9,2,8,1,6],[2,1,7,3,4,6,5,9,
8],[8,5,6,1,2,9,4,7,3],[9,4,3,7,5,8,1,6,2],[1,7,4,5,6,3,2,8,9],[6,2,3
,8,9,7,1,5,4],[9,5,8,2,1,4,7,6,3],[2,9,6,4,8,5,3,1,7],[8,3,1,6,7,9,4,
2,5],[7,4,5,1,3,2,6,9,8],[3,6,9,7,2,8,5,4,1],[4,8,2,3,5,1,9,7,6],[5,1
,7,9,4,6,8,3,2],[1,6,9,7,2,5,4,3,8],[2,8,7,9,3,4,6,1,5],[3,4,5,6,8,1,
9,2,7],[5,8,2,6,9,1,3,7,4],[4,6,1,8,7,3,5,9,2],[7,3,9,2,5,4,8,1,6],[2
,1,7,8,5,6,9,4,3],[3,4,6,1,2,9,7,5,8],[5,9,8,4,7,3,1,6,2]]
```

```
*Main> cells exampleRight
[[1,6,9,7,2,5,4,3,8],[2,8,7,9,3,4,6,1,5],[3,4,5,6,8,1,9,2,7],[5,8,2,6
,9,1,3,7,4],[4,6,1,8,7,3,5,9,2],[7,3,9,2,5,4,8,1,6],[2,1,7,8,5,6,9,4,
3],[3,4,6,1,2,9,7,5,8],[5,9,8,4,7,3,1,6,2]]
```

Let us look at these examples a bit closer to understand them. The function "xall" returns the list of all rows, all columns and all cells in one big list and makes use of the function "cells" to actually get the list for all cells. The table below shows the list of rows, one row on top of each other with indications on the columns and cells and allows us to prove that the lists returned by "xall" and "cells" are correct.

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

Fig.26. Example

5 Future Development

In this chapter I want to discuss some of the things I could implement, change or add in any future development of my program. This project has much potential. There are many topics that could be deepened and extended. This includes some of the theoretical aspects as well as some practical things. I will divide all these ideas into three areas; bugs, improvements and extensions.

5.1 Bugs

1. The 0 problem

This is obviously one of the more serious issues that should have been addressed already. Ideas of how to solve this problem or indeed work around it could be to add some if statements and in case an entry field is empty we mechanically add a 0 to the corresponding list.

5.2 Improvements

2. Lock given numbers in grid
3. Nicer GUI, bigger font
4. Pencil function
5. Improve Brute Force Solver with heuristics

These are some general improvements for the game. Being able to lock the initial set of numbers is a convenient feature and can probably be implemented by making better use of the Glade program and the attributes for each text entry field. Similarly, making the GUI look a bit nicer and increase the font size for example should be doable with Glade.

Some more advanced improvements could be to add the possibility of writing small notes in the corner of each text entry field. I am not sure how we could do this; I guess one way would be to add a button which activates the “pencil” function which then just simply puts some label on top of the text field maybe. But I don’t know Glade enough to make guesses on what is possible and what not. It is defiantly an interesting feature to add, both for the puzzle experience and also as a challenge for the coder.

The other big improvement is of course the Brute Force Solver. Adding some clever heuristics to make it faster so it can actually solve all kinds of puzzles just as quick as the SAT Solver would be great. There are plenty of heuristics and solving techniques that could be integrated into the solver.

5.3 Extensions

6. Colours/shapes/letters
7. Different grid sizes
8. Creating puzzles
9. Implementing other SAT Solvers
10. Complexity discussion (e.g. Heuristics vs. SAT)
11. Implement SAT Solver with other puzzles?

The possible extensions to this program are many. First of all it would be nice to add some variations to the game. For example the integration of colours, using shapes or pictures instead of numbers or even different alphabets. Also interesting would be to add different grid sizes. But this feature would require many changes in how my functions are written. Many of them use lists of ints for example and this would need to be changed then. It would probably be possible by adding polymorphism to those functions but the 0 problem needs to be taken into account as well.

Creating Sudoku puzzles can prove very difficult and is definitely a major task. It is easier said than done really, there are many things to be aware of when creating a puzzle. Standard Sudokus are somewhat symmetric and of course only yield to one unique solution. We need to make sure these requirements are met and well, also that the puzzle is in fact solvable by a human player and not too hard. Another factor is randomness. How do we make sure we generate truly random puzzles each time and not just the same one over and over?

The next possible feature to add is a different SAT Solver. We could even add several different ones in order to compare their performance. This would require some changes in the code of course. We would need to adopt the input and output file handling for example. Not all SAT Solvers use the DIMACS CNF format, or produce a slightly different output file. The code I have at the minute is specifically written to deal with MiniSats output.

A rather theoretical subject but very interesting is the complexity of the code for the Brute Force Solver and the SAT Solver. We could study the performance of both Solvers in much more details and produce discussions on time and space aspects. Especially of course once we have added some more heuristics to the Brute Force Solver. Comparing them will yield to many questions. Which one is really faster? How do they compare on really difficult puzzles or on larger puzzles like a 32x32 grid? Can we integrate them into a puzzle generator?

A last thought is also to try and add some other logic puzzles to the game program. Examples could be Slither Link or Cross Sums. It should be easy to translate these puzzles to CNF format so we can solve them using our SAT Solver. Obviously we could also try and write Brute Force Solvers for these puzzles in addition to that. But this is really far in the future.

6 Critical Appraisal

My project has quite evolved a bit over the time, especially the first few weeks. It seemed there was the general idea of doing some sort of Sudoku program but it ended up being something I would have never imagined to be honest. I did bring a lot of motivation into the project to start with but I lost it pretty soon after I realised how difficult it was to get started with the whole SAT idea, which wasn't even mentioned in the original project proposal [32], and how little Haskell I really knew. But before I go into much detail I will try and organise the things I want to say about this last year chronologically in the next chapter.

6.1 Summary of Completed Work

I have managed to write a Sudoku program that can solve puzzles entered by the user using one of two solver options. One is a very simple and slow Brute Force Solver and the other is the SAT Solver MiniSat. All of my code, including the GUI and the Brute Force Solver, has been written in Haskell. The program also gives the opportunity to load two example puzzles for demonstration purposes. It provides a button to clear the grid and to check if the current state of the puzzle is valid according to the rules or not. A conversion program for the SAT pre-processing is integrated into the game (the translation from the puzzle to the DIMACS CNF format and the other way around).

All objectives as described in the first chapter have been met.

6.2 What went well and what went badly?

I will divide the main areas or aspects of my project into several parts so I can discuss each of them in detail.

6.2.1 Haskell & GUI

Writing a graphical user interface in Haskell is different. It is probably not the best choice to use a functional programming language for the development of nice interfaces. Nevertheless it was a choice I made and I did go through with it. One of the reasons I wanted to use Haskell as the main programming language in my project was that we didn't learn it as much as Java. I wanted to teach myself more about the language and actually use it for a bigger project, a real program instead of only simple small functions we did in the second year course. Also, we were never really taught how to create GUIs so I thought it would be a nice challenge to try and actually do it in Haskell, although I wasn't even sure if it's possible at all.

After some research and testing I found a library I liked. Gtk2Hs also supported Glade which seemed like a useful tool. After using it now though I have to say I could have probably found something better. Glade is a powerful tool to create complex user interfaces but it is not very well documented and there are as good as no tutorials on how to use it with Gtk2Hs. In fact this library itself isn't documented well. This lack of help made it very difficult for me to understand how the program works or how to use it to its full capabilities. Many areas of my GUI could have been improved if I knew how to handle Glade or Gtk2Hs better.

Many of the features I originally wanted to add haven't been implemented in the end due to the problems I had with Glade and the GUI library in general. For example the feature of adding small notes to mark possible values for a field has not been coded. Or the simple fact of the small font size is another good example of something that went very bad. I wanted to increase the size of the font inside the text fields of the grid so it is a bit more readable, but I just didn't find a way to do it in Glade and it seems I would need to make use of yet another library called Pango [30]. Because I had several other problems to fix and most of all other priorities, I left this matter for future development.

6.2.2 Brute Force Solver

The whole aspect of the Brute Force Solver can be summarised best in one word: disaster. I spend way too much time working on it and I still haven't got a decent fast solver in the end. I had troubles implementing the simplest ideas in Haskell for some time, especially during the first semester. At the same time I needed to do all my research on SAT and it all seemed a bit desperate. So I spent a lot of my time reading up on Haskell and experimented a lot with different approaches of how to deal with a Brute Force Solver.

Most of my initial problems were how to handle the values in the grid. Should I save them in a list? A list of tuples? A list of ints? A matrix? I played around with everything. Haskell is a very nice language for lists but a very bad one for variables. One of my problems was that I got stuck with the idea of imperative programming styles. I wanted to use loops and go recursively through a list and update it as it comes. Haskell doesn't allow you to do something like this. Variables can't be updated or changed. Not in that sense anyway. So I needed some other way of doing accomplishing this task. In the end I was stuck with list comprehensions and decided to go with it. No matrix, no tuples, just plain lists.

So after some time of coding with those lists I had a solver.

HOWEVER, it was slow.

So slow that I couldn't even test my solver. The reason for this was that I created a list of all possible combinations of valid Sudoku grids. That is a list with the following number of values inside: 6,670,903,752,021,072,936,960 (See [10]). I went through this list then and compared the given values of the current grid with the corresponding ones in each possible Sudoku solution. This theoretically leads to a solution but it just takes an unreasonable amount of time to find that one solution to the given puzzle in this long list. Therefore I needed to change something drastically. I didn't want to use the heuristics from any existing Sudoku solvers written in Haskell so I played around with my code a while longer.

After a while of desperately trying to find a good way to make this quicker I had the idea of trying to add additional constraints to the list comprehensions I was using.

So to find the solution(s) to one single row to start with, I did the following changes. Instead of defining a list of all possible row combinations and then going through that list and comparing each possibility with my current row, I put some more restrictions on the resulting list directly while creating it in the list comprehension.

```
rows :: Row -> [Row]
rows [i1,i2,i3,i4,i5,i6,i7,i8,i9] =
      [[a,b,c,d,e,f,g,h,i] | a <- (if (i1 == 0) then [1..9] else [i1]),
                             b <- (if (i2 == 0) then [1..9] else [i2]),
                             c <- (if (i3 == 0) then [1..9] else [i3]),
                             d <- (if (i4 == 0) then [1..9] else [i4]),
                             e <- (if (i5 == 0) then [1..9] else [i5]),
                             f <- (if (i6 == 0) then [1..9] else [i6]),
                             g <- (if (i7 == 0) then [1..9] else [i7]),
                             h <- (if (i8 == 0) then [1..9] else [i8]),
                             i <- (if (i9 == 0) then [1..9] else [i9]),
                             Check.check [a,b,c,d,e,f,g,h,i]]
```

These if statements inside the list comprehension are what saved my life ☺
It combines the creation of all possible rows and at the same time filtering out all the ones that don’t fit the given values of the current row.

The solver as it is now, using the function explained above, is good enough to solve very very simple puzzles but still takes endlessly long to solve difficult ones or even normal ones. This is disappointing in a way but enough for the purpose of this project. The disaster turned out to produce something good in the end at least.

6.2.3 Translating Sudoku into DIMACS CNF

This task had its easy and difficult parts I guess. Of course I had to do a lot of research on it. Actually it took me a while to find out that there is a certain format required in order to use a SAT Solver and what that format is. My knowledge of propositional logic had almost faded completely so I needed to do a lot of reading to get into it again. Conjunctive normal form is pretty simple but it is also a very abstract idea to translate a Sudoku puzzle into it. How to translate a puzzle into logic? That’s something my mind had to get used to first I think.

After a lot of reading and confusion I found out it is quite simple. I just needed to come up with the variables for each possible value inside each field. That’s 81 fields times 9 possible values. Right. Easy. 729.

Then I just had to think about how to describe the rules of the game into logic. But that was quite simple too in the end. Conjunctive normal form. Remember?

So it’s a conjunction of clauses and each clause is a disjunction of variables.

```
(X or Y) and (Z or V or W) ……
```

This basically means I can really literally translate the Sudoku rules into this form. Just look at the following sentence.

(Either there is a 1 in field (1,1) ***or*** *there is a 2 in it* ***or*** *a 3* ***or*** *a 4…..* ***or*** *a 9)* ***and***
(there’s a 2 either in field (7,1) ***or*** *in (7,2)* ***or*** *(7,3)……* ***or*** *in (7,9))*

The direct translation of these two English sentences into DIMACS CNF is this:

```
111 112 113 114 115 116 117 118 119 0
712 722 732 742 752 762 772 782 792 0
```

Simple. Just a question of writing it all out without making any stupid mistakes and not forgetting anything.

6.2.4 Generating the SAT input file

The creation of the input file in DIMACS CNF format was challenging. I have explained in detail how I managed to do so in chapter 4.3.2.1 though. I wasn't sure how to handle it all at the beginning. The input file consists of three parts, the first line which varies in every file, the description of the Sudoku rules which is the same for every file and the given values inside the grid which obviously change every time.

The Sudoku rules for example are about 12000 lines long. This is a bit too long to just save in a string somewhere in my code. So I decided to put it in a separate file and access it every time I need to only. Reading and writing to files in Haskell is surprisingly easy. So far so good. I construct the first line, add the rules which I read in from some template file and then add the last few lines with the current values of the puzzle. Tada, there's my input file.

Same goes for reading in the output file and extracting the valid values for each field. Easy done.

6.2.5 Integration of MiniSat

The most notable thing that went very well in my project was the integration of the SAT Solver. After months of research and lots of worrying I actually sat down and tried to find out how to use MiniSat. It turned out to be fairly easy. Compile in Cygwin and execute with two arguments. Maybe that's because MiniSat is supposed to be very suitable for beginners in the SAT community but maybe not and I was just really clever ☺

I didn't find any tutorial or guideline on how to use SAT Solvers anywhere. So even though it was quite simple in the end it made me worry a lot at first. SAT seems to be a technology still being researched a lot. It seems powerful but underestimated maybe. I had troubles understanding the usage and how to integrate it into my program. It might sound silly, but it is not something we were ever taught. How do you call a program from within some other? Well this is just another thing I had to look into.

Anyway, Haskell has a sweet little function called "system" which passes the command to the Windows command interpreter. This means I could just save the following line in a string and pass it as an argument to the function "system".

```
cmd = "minisat cnf/input.cnf cnf/output.cnf"

system cmd
```

This is it. Really simple. I just call MiniSat with two arguments; the input file and the output file.

6.3 What would I do differently now?

What I'd do differently if I could start all over? I am not sure I would change things.

I just wish I had managed to get this Brute Force Solver to work sooner. Because then I would have been able to start earlier with the whole SAT Solver integration and I would have had more time to improve the Brute Force and to add more features to the GUI and so on. It just was all delayed a lot because of these problems I had during the end of the first semester and the beginning of the second. But it is always easy to say I would have structured my work better or studied harder. Actually doing it is another story.

Overall I think this project turned out to be ok. I had some troubles in between but in the end I managed to get basically everything done I wanted to. It was very rewarding for me as I have learned a lot of new things about Haskell and SAT.

7 Bibliography and Citations

[1] http://en.wikipedia.org/wiki/Sudoku - Wikipedia about “Sudoku”

[2] http://en.wikipedia.org/wiki/Boolean_satisfiability_problem - Wikipedia about “Boolean satisfiability problem”

[3] http://www.haskell.org/haskellwiki/Haskell - Haskell Wiki

[4] http://haskell.org/gtk2hs/ - Gtk2Hs

[5] http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/ - MiniSat

[6] http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps - DIMACS CNF *“Satisfiability Suggested Format*”, Paper from 1993

[7] http://www.nikoli.co.jp/en/puzzles/sudoku/index_text.htm - Nikoli

[8] http://en.wikipedia.org/wiki/Latin_square - Wikipedia about “Latin Square”

[9] http://people.csse.uwa.edu.au/gordon/sudokumin.php - Minimum Sudoku by Gordon Royle

[10] http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf - Bertram Felgenhauer, Frazer Jarvis, „*Mathematics of Sudoku*“ Paper from 2006

[11] http://en.wikipedia.org/wiki/NP-complete - Wikipedia about “NP-complete”

[12] http://en.wikipedia.org/wiki/Main_Page - Wikipedia, The Free Encyclopedia

[13] http://www.cs.toronto.edu/~sacook/homepage/1971.pdf.gz - Stephen Cook, “*The Complexity of Theorem Proving”,* Paper from 1971

[14] http://en.wikipedia.org/wiki/Conjunctive_normal_form - Wikipedia about “Conjunctive normal form”

[15] http://www.satcompetition.org/ - SAT competitions

[16] http://en.wikipedia.org/wiki/DPLL_algorithm - Wikipedia about “DPLL Algorithm”

[17] http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html - Ivor Spence, The Sudoku Puzzle as a Satisfiability Problem

[18] http://www.sat4j.org/ - Java Satisfiability Library

[19] http://www.cril.univ-artois.fr/~roussel/satgame/satgame.php?level=1&lang=eng – The SAT Game

[20] http://en.wikibooks.org/wiki/Haskell - WikiBook on Haskell

[21] http://haskell.org/hoogle/ - Hoogle

[22] Simon Thompson, *"Haskell: The Craft of Functional Programming"*, Addison-Wesley, ISBN: 0-201-34275-8

[23] Paul Hudak, *"The Haskell School of Expression: learning functional programming through multimedia",* Cambridge University Press, ISBN: 0-521-64408-9

[24] http://www.haskell.org/haskellwiki/Sudoku - Collection of Haskell Sudoku Solvers

[25] http://en.wikipedia.org/wiki/Rational_Unified_Process - Wikipedia about "Rational Unified Process"

[26] http://www.haskell.org/ghc/ - Glasgow Haskell Compiler

[27] http://glade.gnome.org/ - Glade

[28] http://dimacs.rutgers.edu/Challenges/ - DIMACS Challenge

[29] http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html - Haskell User Guide Profiling Chapter

[30] http://www.pango.org/ - Pango Library

[31] http://www.haskell.org/onlinereport/standard-prelude.html - Haskell Standard Prelude

[32] https://campus.cs.le.ac.uk/teaching/resources/CO3015/ProjProposals06.xml#project-63 – Project Proposal for Sudoku Project

8 Appendix

8.1 Code

Here I have included the function definitions from predefined Haskell functions I have used in my code.

8.1.1 Haskell Standard Prelude [31]

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

```
any, all         :: (a -> Bool) -> [a] -> Bool
any p            =  or . map p
all p            =  and . map p
```

```
and, or          :: [Bool] -> Bool
and              =  foldr (&&) True
or               =  foldr (||) False
```

```
concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

```
drop                   :: Int -> [a] -> [a]
drop n xs      | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs
```

```
elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

```
head             :: [a] -> a
head (x:_)       =  x
head []          =  error "Prelude.head: empty list"
```

```
length            :: [a] -> Int
length []         =  0
length (_:l)      =  1 + length l
```

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
print      :: Show a => a -> IO ()
print x    =  putStrLn (show x)
```

```
readFile   :: FilePath -> IO String
readFile   =  primReadFile
```

```
sequence       :: Monad m => [m a] -> m [a]
sequence       =  foldr mcons (return [])
                    where mcons p q = p >>= \x -> q >>= \y -> return (x:y)
```

```
tail              :: [a] -> [a]
tail (_:xs)       =  xs
tail []           =  error "Prelude.tail: empty list"
```

```
take                    :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

```
words             :: String -> [String]
words s           =  case dropWhile Char.isSpace s of
                       "" -> []
                       s' -> w : words s''
                             where (w, s'') = break Char.isSpace s'
```

```
writeFile  :: FilePath -> String -> IO ()
writeFile  =  primWriteFile
```

8.1.2 Others

```
-- Character-testing operations
isDigit :: Char -> Bool
isDigit c                =  c >= '0' && c <= '9'
```

```
-- transpose is lazy in both rows and columns,
--       and works for non-rectangular 'matrices'
-- For example, transpose [[1,2],[3,4,5],[]]  =  [[1,3],[2,4],[5]]
-- Note that [h | (h:t) <- xss] is not the same as (map head xss)
--      because the former discards empty sublists inside xss
transpose                :: [[a]] -> [[a]]
transpose []             = []
transpose ([]     : xss) = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:t) <- xss]) :
                             transpose (xs : [t | (h:t) <- xss])
```

```
{- |If the operating system has a notion of current directories,
@'setCurrentDirectory' dir@ changes the current
directory of the calling process to /dir/.
The operation may fail with:

* 'HardwareFault'
A physical I\/O error has occurred.
@[EIO]@

* 'InvalidArgument'
The operand is not a valid directory name.
@[ENAMETOOLONG, ELOOP]@

* 'isDoesNotExistError' \/ 'NoSuchThing'
The directory does not exist.
@[ENOENT, ENOTDIR]@

* 'isPermissionError' \/ 'PermissionDenied'
The process has insufficient privileges to perform the operation.
@[EACCES]@

* 'UnsupportedOperation'
The operating system has no notion of current directory, or the
current directory cannot be dynamically changed.

* 'InappropriateType'
The path refers to an existing non-directory object.
@[ENOTDIR]@

-}

setCurrentDirectory :: FilePath -> IO ()
setCurrentDirectory path = do
  modifyIOError (`ioeSetFileName` path) $
    withCString path $ \s ->
       throwErrnoIfMinus1Retry_ "setCurrentDirectory" (c_chdir s)
        -- ToDo: add path to error
```

```
{-|
Computation @system cmd@ returns the exit code
produced when the operating system processes the command @cmd@.

This computation may fail with

* @PermissionDenied@: The process has insufficient privileges to
perform the operation.

* @ResourceExhausted@: Insufficient resources are available to
perform the operation.

* @UnsupportedOperation@: The implementation does not support
system calls.

On Windows, 'system' is implemented using Windows's native system
call, which ignores the @SHELL@ environment variable, and always
passes the command to the Windows command interpreter (@CMD.EXE@ or
@COMMAND.COM@), hence Unixy shell tricks will not work.
-}
ifdef __GLASGOW_HASKELL__
system :: String -> IO ExitCode
system "" = ioException (IOError Nothing InvalidArgument "system" "null
command" Nothing)
system str = do
#if mingw32_HOST_OS
```

```
p <- runCommand str
waitForProcess p
#else
-- The POSIX version of system needs to do some manipulation of signal
-- handlers. Since we're going to be synchronously waiting for the child,
-- we want to ignore ^C in the parent, but handle it the default way
-- in the child (using SIG_DFL isn't really correct, it should be the
-- original signal handler, but the GHC RTS will have already set up
-- its own handler and we don't want to use that).
old_int <- installHandler sigINT Ignore Nothing
old_quit <- installHandler sigQUIT Ignore Nothing
(cmd,args) <- commandToProcess str
p <- runProcessPosix "runCommand" cmd args Nothing Nothing
Nothing Nothing Nothing
(Just defaultSignal) (Just defaultSignal)
r <- waitForProcess p
installHandler sigINT old_int Nothing
installHandler sigQUIT old_quit Nothing
return r
#endif /* mingw32_HOST_OS */
#endif /* __GLASGOW_HASKELL__ */
```

8.1.3 Original Project Proposal [32]

63. Sudoku

Supervisor: Fer-Jan de Vries

Prerequisites
Enthusiasm and interest to get involved in the topic.

Aims of Project
Recently Sudoku has recently become very popular outside Japan. Sudoku puzzles appear daily in most newspapers. Perhaps Sudoku was invented in Japan to have a counterpart for the crosswords which don't work well in their character script. There are other interesting Japanese puzzles as well. Make a tool that allows to solve such puzzles online using mouse and keyboard. Perhaps such a tool can not only solve such puzzles but even better give hints in case the human problem solver gets stuck. Another challenging task is to make a program that produces such puzzles.

Challenges presented by the project
It is a challenge to write a program that can give hints to a human problem solver. puzzles. It is another challenge to write a program that actually designs or helps with the design of proper puzzles.

Learning Outcomes
A great experience in problem solving and programming. You will learn/apply some AI.

Nature of End-Product
A package for creating Japanese Garden Puzzle, and solving them both interactively and automatically. An account of the algorithms for correct move checking and the puzzle solving.

Project Timetable

Semester1
Research on graphics and algorithms for solving the Puzzle, design of package and implementation of the Puzzle Inputter, Puzzle Solver Assistant and Puzzle Editor.
Semester2
Design and implementation of the Puzzle Designer and Puzzle Solver.

References

Matt Ginsberg.
Essentials of Artificial Intelligence. Morgan Kaufmann Publishers (1993).
Anonymous.
I learned about these puzzles via a Japanese puzzlebook, published in 1993 by the Japanese publisher Nikoli (tel: +88 3 3485 2081). It has no ISBN number...

8.2 Diaries