FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Lot: Prototyp eines constraints-basierten Spreadsheets

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Esad Hajdarevic
Matrikelnummer 0347783

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Wien, 4. April 2016

Esad Hajdarevic	M. Anton Ertl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Lot: prototype of a constraints-based spreadsheet

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Esad Hajdarevic
Registration Number 0347783

to the Faculty of Informatics
at the TU Wien
Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

Vienna, 4th April, 2016

Esad Hajdarevic	M. Anton Ertl

Erklärung zur Verfassung der Arbeit

Esad Hajdarevic
Seidengasse 32/3/56, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. April 2016

Esad Hajdarevic

Kurzfassung

Die Berechnungsregeln von Spreadsheets können als Sammlung von Constraints aufgefasst werden, die nur in eine Richtung ausgewertet werden. Oft wäre es vorteilhaft, wenn auch die andere Richtung verwendet werden könnte (z.B.: vorgegebene Gesamtsumme, wobei einige Posten noch variiert werden können). Denn nicht selten führt die festgelegte Richtung der Auswertung zu einer zeitraubenden Versuch-und-Irrtum-Arbeitsweise oder zur Eingabe redundanter (und möglicherweise falscher) Berechnungsregeln.

Um dieses Problem im Detail zu betrachten und zu bearbeiten, habe ich im Rahmen dieser Bachelorarbeit ein web-basiertes Spreadsheet erstellt, das die Berechnungsregeln in alle Richtungen verwendet, bzw. ein constraint-basiertes Arbeiten ermöglicht. Anhand mehrere Beispiele wird gezeigt, dass durch diese Erweiterung Spreadsheets auch für Probleme eingesetzt werden können, für die sie sonst ungeeignet wären.

Abstract

Spreadsheet formulas can be seen as a collection of unidirectional constraints. Users often want to apply the formulas in the other direction (for example, to find values that add up to a certain sum vs. calculating the sum), which can lead to a time consuming trial-and-error approach of finding the right values or error-prone rewriting of existing formulas.

As part of this thesis, I developed a web-based spreadsheet that supports working with constraints. Number of examples are presented to illustrate that addition of constraints make spreadsheets applicable to a whole new set of problems for which they would otherwise be unsuitable.

Contents

CHAPTER 1

Introduction

Traditional spreadsheets can be seen as a collection of cells, arranged in a two-dimensional matrix, that contain either a value or a formula. A value represents what is actually displayed in a spreadsheet (a string, type of number, date etc.)[1] and a formula describes the value of a cell by transforming values of other cells or constants.

As a cell cannot simultaneously contain both a value and a formula, this makes formulas inherently unidirectional [MPSC14]. For instance, a cell C holding the formula $= A+B$ computes the value of C from the values of A and B, but the user is not able to compute the values of A and/or B by editing C.

This makes modeling some quite simple problems in a traditional spreadsheet relatively cumbersome. Let us consider one such example used to calculate fictional sale profits shown in fig. 1.1.

Determining how many more units need to be sold for profits to double cannot be done by simply doubling the value in cell *B5*, as it would erase the previously defined formula that established the relationship between the values in the first place.

In this case, users usually resort to what [Lel88] calls a "manual relaxation", where various values for *B1* (Units sold) are tried until *B5* (Profit) shows the right value. Alternatively, users can rewrite the formula to *=(B5+B4)/(B3-B2)*, which assigns a value to *B1* in terms of *B5*. However, as the relationships between cells in a spreadsheet get more complex, such formula rewriting approach becomes increasingly non-trivial and error prone.

The basic idea of a constraints-based spreadsheet is to consider both values and formulas to be constraints—equality relationships between the cell they are defined in on one side, and other cells or constants on the other. By using a constraint-satisfaction mechanism

[1] Empty cell and an error resulting from a formula that can not be evaluated can be considered as special kind of value

	A	*B*
1	Units sold	1000
2	Material cost	5
3	Unit price	10
4	Fixed costs	2000
5	Profit	= (B3-B2)*B1 - B4

	A	*B*
1	Units sold	1000
2	Material cost	5
3	Unit price	10
4	Fixed costs	2000
5	Profit	3000

Figure 1.1: A spreadsheet used to calculate fictional sale profits[2]

(solver) we can then determine the values for cells that satisfy these relationships. Multiple such constraints can be defined for one cell (e.g. $C = A+B$; $C = 400$) and in addition to equality, other relationships such as $>$, $<$ or $\neq$ between cells can be expressed.

This not only achieves bidirectionality of formulas and lets us easily solve calculations like the one described above, but also makes it possible for such system to solve whole class of "guessing" problems, as we will see in the presented examples.

As part of this thesis, I implemented a prototype of such spreadsheet ontop of the Z3 solver [Mic] called Lot[3], to explore how the traditional spreadsheet UI can be extended to support working with constraints.

[2]Samples will be presented in this form, with user input on the left and an evaluated spreadsheet on the right

[3]Available online at `http://try-lot.appspot.com/`, source code at `http://github.com/esad/Lot`

CHAPTER 2

Lot

2.1 Overview

At first glance, Lot looks like a traditional spreadsheet without formatting and other tools that were outside of the scope of the prototype (Figure 2.1). Selection and editing of cells work like expected: arrow keys or mouse can be used to change the selection, double-clicking or pressing the enter key while a cell is selected lets user edit the cell contents inline. Some other basic spreadsheet ergonomics are implemented, for example after pressing ⏎ to finish editing of a cell, the selection advances into the next row, supporting rapid input of series of values.

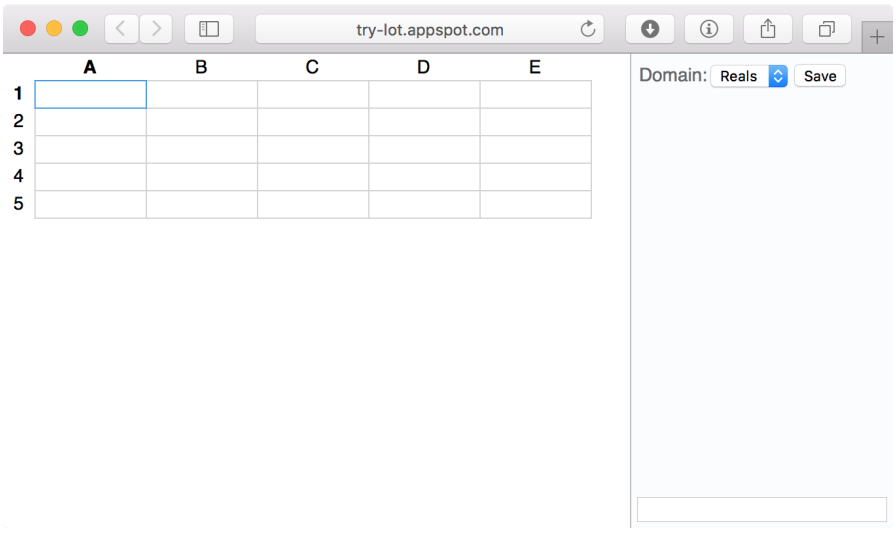

Figure 2.1: Initial screen of Lot

One visible difference is the addition of the constraints sidebar, which gives an overview of all constraints defined in a spreadsheet. Here, the user can also add new constraints over multiple cells, for example *A1+A2 = B4-B3*.

After entering the spreadsheet from fig. 1.1 into Lot we obtain the same initial result. To answer the question of how many items need to be sold for profits to double, we can now simply remove the value in *B1* and enter an additional value constraint *= 6000* into *B5* (Figure 2.2). Derived solution will be shown in *B1*:

	A	*B*
1	Units sold	
2	Material cost	5
3	Unit price	10
4	Fixed costs	2000
5	Profit	= (B3-B2)*B1 - B4; = 6000

	A	*B*
1	Units sold	1600
2	Material cost	5
3	Unit price	10
4	Fixed costs	2000
5	Profit	6000

Figure 2.2: Fictional sale profits spreadsheet in Lot, with an additional constraint on profits

Note that if we try to supply a different value for *B1*, the set of constraints becomes unsatisfiable and the following error message is shown:

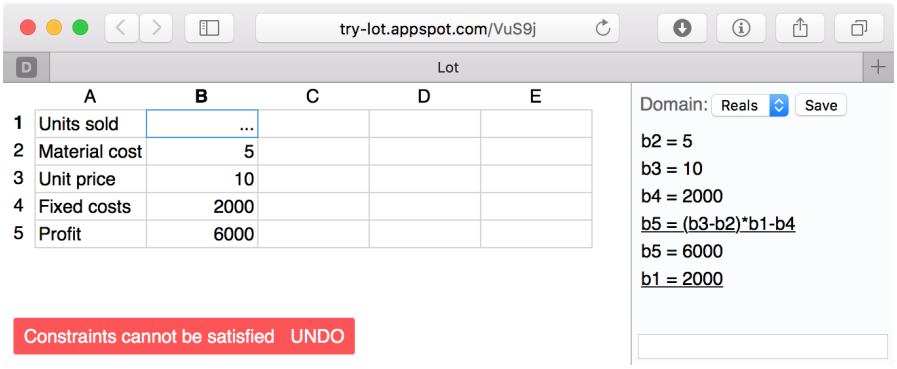

Figure 2.3: Lot displaying a message that defined constraints cannot be satisfied

Currently, Lot is uncapable of detecting which constraints are in conflict. However, as working in a spreadsheet is inherently incremental (cell values and constraints are entered in a sequence), showing the error message as soon as a conflicting set of constraints is detected is usually helpful enough for user to spot the cause of unsatisfiability.

2.2 Constraints

Internally, a constraint in Lot is described by the following algebraic data types[1]:

```
type Constraint = Constraint Expr Rel Expr
                | PredicateConstraint Predicate (List Expr)
type Expr = Const Float | Id String | Calc Op Expr Expr
type Rel = Eq | NotEq | Lt | LtEq | Gt | GtEq
type Op = Add | Sub | Mul | Div
type Predicate = Distinct
```

Figure 2.4: ADTs describing the abstract syntax of a constraint in Lot

We distinguish between *relational constraints* and *predicate constraints*. A relational constraint asserts a relationship between two expressions, described by a relational operator such as $=$ or $>$. Predicate constraint asserts a relationship between multiple expressions, described by the predicate itself. Currently, only one such predicate named *distinct* is implemented. It asserts that all supplied expressions must have a different value. We can use it to write *distinct(A1,A2,A3,A4,A5)* instead of *A1 != A2; A2 != A3; A3 != A4; A4 != A5*.

Constraints are entered using a syntax similar to the one for formulas in a traditional spreadsheet. They can be entered either directly in a cell (for example, by typing *=(A1+A2)/2* into *A3*) or in the sidebar. When constraints are entered in the cell, a left-side expression `Id <cell-address>` is assumed. In the sidebar, the left side of the relation cannot be inferred and must be specified. Multiple constraints can be entered in one line, separated by `;`.

When user edits a cell, all the constraints with the left side of `Id <cell-address>` are converted to their textual representation, with left-side expression omitted. The textual representations are then concatenated with the delimiter `;` and shown to the user for editing.

Predicate constraints can currently only be entered in the sidebar. Future extensions may include unary predicates such as *maximize* that would be meaningful in the context of a single cell. Users could then enter *!maximize* in a cell to denote that the value of that cell should be maximized.

[1] Listing taken directly from the implementation in the Elm programming language

CHAPTER 3

Examples

The following examples will show how Lot can be used to solve some problems beyond usual spreadsheet calculations.

3.1 Temperature converter

In the first example, we will write a simple temperature converter between Celsius degrees (C), Fahrenheit degrees (F) and Kelvin (K), as described by the following equations:

$$C = (F - 32) * 5/9$$
$$K = C + 273.15$$

As discussed in chapter 1, in order to convert between any two units in a conventional spreadsheet, we would have to derive 6 separate formulas ($C \to F, F \to C, C \to K, K \to C, F \to K, K \to F$). We can express this more naturally in Lot, by having only two constraints in the cells holding Celsius and Kelvin values:

	A	*B*
1	Celsius	= (B2-32)*5/9
2	Fahrenheit	
3	Kelvin	= B1+273.15

	A	*B*
1	Celsius	0
2	Fahrenheit	32
3	Kelvin	273.15

Figure 3.1: Spreadsheet for a temperature converter

Even though we didn't supply any values, Lot will supply initial values that satisfy the constraints. To convert beetween temperatures, we just need to add one additional value

constraint to the source cell. For example, to convert from 451 °F, we just need to add an additional *B2 = 451* constraint:

	A	*B*
1	Celsius	= (B2-32)*5/9
2	Fahrenheit	= 451
3	Kelvin	= B1+273.15

	A	*B*
1	Celsius	232.77777
2	Fahrenheit	451
3	Kelvin	505.92777

Figure 3.2: Spreadsheet for a temperature converter with Fahrenheit input

We can also check for which temperature the Celsius and the Fahrenheit scale will have the same value:

	A	*B*
1	Celsius	= (B2-32)*5/9
2	Fahrenheit	= B1
3	Kelvin	= B1+273.15

	A	*B*
1	Celsius	-40
2	Fahrenheit	-40
3	Kelvin	233.15

Figure 3.3: Spreadsheet for finding the temperature which has same Celsius and Fahrenheit readings

3.2 Magic squares

An order-n magic square is a $n \times n$ matrix containing numbers 1 to n^2, with each row, column and main diagonal having the same sum. In this example we will use Lot to find a 3×3 magic square.

It should be noted that identifiers in constraints are not limited to cell references such as *A1*, but can be arbitrary labels. We can use one such identifier to denote the sum that each row, column and main diagonal should have, and use it to simplify expressing the required constraints.

In this example, we will use sidebar to enter global constraints and make use of the *distinct* predicate to assert that all cells should have different values. After entering the constraints we arrive at the result shown in fig. 3.4.

As magic square is defined as sequence of natural numbers, we will further need to switch the *Domain* to *Ints* in the sidebar to tell the solver to consider only integer solutions, but this will also include negative numbers. Unfortunately, Lot currently doesn't implement an universal quantifier nor a predicate that would let us concisely limit the range of values, so we'll have to supply these manually. We arrive at the final solution in fig. 3.5.

A1+A2+A3 = S
B1+B2+B3 = S
C1+C2+C3 = S
A1+B1+C1 = S
A2+B2+C2 = S
A3+B3+C3 = S
A1+B2+C3 = S
A3+B2+C1 = S
distinct(A1,A2,A3,B1,B2,B3,C1,C2,C3)

	A	*B*	*C*
1	-0.5	0.375	-0.625
2	-0.375	-0.25	-0.125
3	0.125	-0.875	0

Figure 3.4: Almost a magic square

Domain = Ints
A1+A2+A3 = S
B1+B2+B3 = S
C1+C2+C3 = S
A1+B1+C1 = S
A2+B2+C2 = S
A3+B3+C3 = S
A1+B2+C3 = S
A3+B2+C1 = S
distinct(A1,A2,A3,B1,B2,B3,C1,C2,C3)
A1 >= 1; A1 <= 9
A2 >= 1; A2 <= 9
A3 >= 1; A3 <= 9
B1 >= 1; B1 <= 9
B2 >= 1; B2 <= 9
B3 >= 1; B3 <= 9
C1 >= 1; C1 <= 9
C2 >= 1; C2 <= 9
C3 >= 1; C3 <= 9

	A	*B*	*C*
1	2	7	6
2	9	5	1
3	4	3	8

Figure 3.5: Magic square

3.3 Map coloring

Map coloring problem concerns itself with coloring a map of territories with n colors, so that no two neighbouring territories have the same color. It has been proven that any map can be colored by at most 4 colors [Gon07]. In this example, we will use Lot to come up with a sample coloring for the map of Austrian states and determine the maximum number of colors needed to color this map.

Figure 3.6 shows a solution for coloring such map with 3 colors. We arrive at this solution by starting with $n = 4$ and decrementing n until "Constraints cannot be satisfied"

message is displayed. The map in fig. 3.7 is obtained by applying the following coloring function:

$$color(n) : \{0,1,2\} \rightarrow \{red, green, blue\} = \begin{cases} red & \text{if } n = 0 \\ green & \text{if } n = 1 \\ blue & \text{if } n = 2 \end{cases}$$

	A	B
1	W	
2	NÖ	
3	B	
4	St	
5	OÖ	
6	S	
7	K	
8	T	
9	V	

Domain = Ints
b1 != b2
b2 != b4; b2 != b5;
b3 != b2; b3 != b4
b4 != b5
b6 != b5; b6 != b4; b6 != b7
b8 != b6; b8 != b7; b8 != b9
b1 >= 0; b1 < n
b2 >= 0; b2 < n
b3 >= 0; b3 < n
b4 >= 0; b4 < n
b5 >= 0; b5 < n
b6 >= 0; b6 < n
b7 >= 0; b7 < n
b8 >= 0; b8 < n
b9 >= 0; b9 < n
n = 3

	A	B
1	W	1
2	NÖ	0
3	B	2
4	St	1
5	OÖ	2
6	S	0
7	K	2
8	T	1
9	V	0

Figure 3.6: Coloring of Austrian states for n=3

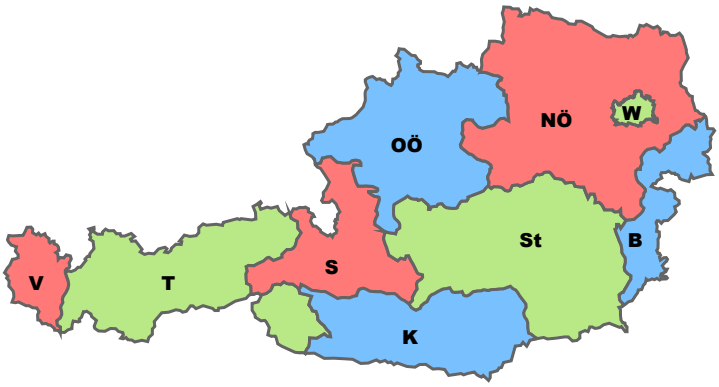

Figure 3.7: Coloring result (Graphics derived by the author from original map by Andreas Griessner, published under the GFDL license)

CHAPTER 4

Implementation

4.1 Initial considerations

I considered the following three approaches for prototype implementation:

1. Extending an existing desktop-based spreadsheet application such as Microsoft Excel, LibreOffice Calc or web-based Google Sheets, all of which include a possibility for developers to extend the built-in functionalities by invoking code defined in an external module. An obvious advantage of such approach is getting the traditional spreadsheet functionality "for free". Familiarity of users with the interface and broad user base would also make future usability study easier. On the other hand, it was not clear if the extensibility offered by the APIs would be flexible enough to support such fundamental changes and whether integrating an external solver was possible at all. While LibreOffice supports extensions in several languages (Java, Python, C++, LibreOffice Basic), Microsoft Excel and Google Sheets only support extensions written in a specific language (VBA and JavaScript, respectively).

2. Modification of existing open-source spreadsheet application. While this option would allow greater flexibility in implementation and make solver integration more feasible, it also implied a steep learning curve to get familiar with the code base[1]. Ultimatively, this option would also impose a limit on the choice of a programming language.

3. Development of a standalone prototype. While offering most flexibility in both choice of the programming language as well as architecture, this option would also

[1] LibreOffice codebase in February 2016 was more than 7 million lines of code long, according to the data from Open Hub (`https://www.openhub.net/p/libreoffice`), a service that collects metrics on various open-source projects

imply additional effort to implement the basic spreadsheet functions first, before the constraints-based related features could be added on top.

I evaluated the above alternatives in terms of the following criteria:

Low barrier to entry Ability to share progress and receive early feedback from the community and my mentor—implied not only that the prototype should be able to run cross-platform, but it also must not involve complex installation and have no security implications (installing unknown binaries).

Solver availability As implementing own solver was beyond the scope of this thesis, availability of the solver and ability to integrate it into prototype was a must-have.

Flexibility Ability to efficiently implement concepts described in this thesis.

Productivity Finally, I wanted to use a programming language/environment that I felt would be most productive choice for implementing the prototype.

The only platform where the first goal could be fully achieved was a browser-based solution. Desktop-based alternatives meant shipping separate add-ons or binaries for each platform that also bundled the solver together with all security implications of such approach. A notable exception would be usage of Java or Python to implement an extension for LibreOffice Calc in combination with a solver that compiled to JVM or had a Python implementation.

Initial research (usage in the babelsberg-js project[2] and [NON15]) indicated that it was possible to cross-compile the Z3 solver written in C++ (and potentially other solvers) to JavaScript using LLVM-to-Javascript compiler *Emscripten* [Zak11].

I opted to development of a standalone browser-based prototype that would use the Emscripten-compiled Z3 solver, even though an integration into Google Sheets or extension of EtherCalc[3] was also a viable option. My rationale was that the flexibility and productivity benefits of such approach would offset the additional effort needed to implement a basic spreadsheet interface.

For the implementation, I choose the programming language Elm [CC13]. Elm is a modern, functional, statically typed language that compiles to JavaScript and offers well-defined interoperability with existing JavaScript code. Some of the Elm's features, such as the strong static type system, support for ADTs, exhaustive pattern matching, availability of powerful parsing libraries and support for *functional reactive programming* proved essential to timely development of the prototype.

[2] `https://github.com/babelsberg/babelsberg-js/tree/master/z3/emz3`
[3] Open-source web-based spreadsheet (`https://ethercalc.net`)

4.2 Solver

In order to solve constraints and arithmetic expressions found in a typical spreadsheet, as well as some less typical problems shown in Chapter 3, a solver that would be used in Lot needed to fullfil the following criteria:

Support for both real numbers and integers

Support for non-linear arithmetic over both domains

Ability to produce models for satisfiable formulas In addition to knowing whether the constraints can be satisifed, we also need the solver to provide us with set of values (a model) that satisifies them

There are certain known limitations on non-linear arithmetic: it is undecidable over integers, that is, there's no general algorithm for solving non-linear integer equalities [MB09]. While it is decidable over reals, it remains very expensive and for many problems a solver will not be able to provide us with a solution.

For above criteria, SMT solvers looked very promising. SMT (Satisfiability Modulo Theories) [MB09] concerns itself with checking satisfiability of formulas with respect to one or more fixed background theories (such as non-linear arithmetic over reals) which constrain the interpretation of symbols used in the formula. SMT solvers modularly combine special purpose algorithms for each domain with boolean satisfiability.

Z3 [Mic] is a state of the art SMT solver from Microsoft Research available under an open-source MIT license. It was an obvious choice for the implementation, as it fullfils all of the stated criteria and additionally supports other features that could be useful for future development, such as support for boolean arithmetic and optimization. A standard input format for SMT solvers called *SMT-LIB 2* (2 stands for second version of the standard) is used to input the generated SMT problem into Z3. SMT-LIB features a LISP-like syntax designed with ease of generation/parsing in mind.

An example of SMT-LIB 2 input generated for the Temperature Converter example from chapter 3 is shown in fig. 4.1.

4.2.1 Emscripten compilation

Emscripten was used to compile the Z3 solver, written in C++, to JavaScript in order to run it in the browser. Emscripten takes a LLVM (Low Level Virtual Machine) assembly generated by one of the frontends (in case of the Z3, the Clang C++ compiler, but other frontends for various languages exist as well) as an input, performs various optimizations and outputs a low-level, optimizable subset of JavaScript called *asm.js*.

Z3 includes a standalone command-line executable that parses a "high-level" SMT-LIB 2 input from STDIN or a file, as well as a static C library that exposes lower-level primitives.

```
(set-option :pp-decimal true)
(declare-const b1 Real)
(declare-const b2 Real)
(declare-const b3 Real)
(assert (= b3 (+ b1 273.15)))
(assert (= b1 (/ (* (- b2 32) 5) 9)))
(check-sat)
sat
(get-value (b1 b2 b3))
((b1 0.0)(b2 32.0)(b3 (/ 5463.0 20.0)))
```

Figure 4.1: Generated SMT-LIB 2 input for the temperature converter example, Z3 output shown in blue

Calling C functions from JavaScript using Emscripten is possible, but it is relatively cumbersome and requires manual redeclaration of all exported functions. The exposed Z3 C API functions are very low-level and it would require significant effort to generate correct formulas from internal constraint representation in Lot using only these primitives.

With this in mind, I opted for generating SMT-LIB 2 code from internal constraint representation, and using Emscripten-provided filesystem API to store the generated program in a virtual input file. Invoking the Z3 command line executable is then simulated by invoking `callMain()` with constructed arguments array (see fig. 4.2).

A big issue was the size of the generated JavaScript file for the Z3 command line tool. With highest optimization settings (`-O3`), Emscripten generated a 17 MB JavaScript output. On the development machine[4], parsing and evaluating this file in recent versions of Chrome and Firefox took up to 10 seconds, during which the browser was completely unresponsive. This meant that the loading had to be made asynchronous and when Lot is started, user is presented with a loading screen.

Compilation with highest optimization settings caused evaluation of certain SMT programs to segfault. This was probably related to memory alignment issues and non-portable Z3 C++ code, however a thorough investigation of the cause proved to be difficult. I noticed that older versions of Z3 (4.3.x and below) didn't exhibit this problem, so I opted to use an older version of the solver. This version doesn't support optimizations and weak constraints.

While disabling optimization helped mitigate the problem, the size of generated JavaScript grew beyond 100 MB, making the loading process take prohibitively long (around 5 minutes).

[4]Late 2013 MacBook Pro, 2.6 GHz Intel Core i5 with 8 GB RAM

```
var stdout = [];
var stderr = [];
z3em.fs.createDataFile("/", "input.smt2", program, true, true);
try {
  z3em.module.print = function(str) { stdout.push(str); }
  z3em.module.printErr = function(str) { stderr.push(str);}
  z3em.module.callMain(["-smt2", "/input.smt2"])
} catch (exception) {
  console.error("z3-emscripten exception:", exception);
} finally {
  z3em.fs.unlink("/input.smt2");
}
if (stdout[0] == "sat") {
  // ...
} else if (stdout[0] == "unsat") {
  // ...
} else {
  // ....
}
```

Figure 4.2: Invoking Z3 command line tool from JavaScript

CHAPTER 5

Future improvements

Due to a limited scope of my work on this thesis, Lot leaves lot of room for future improvements. Some of the most relevant ones are listed below.

5.1 Enumerating solutions

Often a solution found by the solver is not unique. To implement enumeration of all possible solutions (a set which can be infinite, for example in case of $A1 > 0$), user interface would need to be extended with the ability to go back and forth between solutions. It would also be helpful to inform the user of the opposite fact - whenever a solution found is unique.

Z3 doesn't support returning "next" solution, however a simple workaround could be implemented. The idea is to "block" already examined solutions, by adding a set of inequality constraints for them to the set of original constraints. This way, the solver will either find a new solution or report unsatisfiability, in which case we have enumerated all solutions. The same mechanism could be automatically invoked on the first solution to check its uniqueness.

5.2 Optimization support

In most instances of problems with multiple solutions, the user is interested in a "best" solution, which implies optimization. Optimization involves finding such solutions where certain value is maximized or minimized. As Z3 already has support for optimization, implementing it in Lot would be quite straightforward and would involve extending the constraints syntax with new *minimize* and *maximize* predicates and generating the right SMT-LIB2 assertions.

```
Domain = Ints
A1+A2+A3 = S; B1+B2+B3 = S
B1+B2+B3 = S; C1+C2+C3 = S
A1+B1+C1 = S; A2+B2+C2 = S
A3+B3+C3 = S; A1+B2+C3 = S
A3+B2+C1 = S
distinct(A1,A2,A3,B1,B2,B3,C1,C2,C3)
A1 >= 1; A1 <= 9; A2 >= 1; A2 <= 9
A3 >= 1; A3 <= 9; B1 >= 1; B1 <= 9
B2 >= 1; B2 <= 9; B3 >= 1; B3 <= 9
C1 >= 1; C1 <= 9; C2 >= 1; C2 <= 9
C3 >= 1; C3 <= 9
```

```
Domain = Ints
sum(A1:A3) = s; sum(B1:B3) = s
sum(C1:C3) = s; sum(A1:C1) = s
sum(A2:C2) = s; sum(A3:C3) = s
A1+B2+C3 = s; A3+B2+C1 = s
distinct(A1:C3)
all(A1:C3, >= 1); all(A1:C3, <= 9)
```

Figure 5.1: Constraints used to find a 3x3 magic square rewritten to use ranges and a universal quantifier

5.3 Ranges

In a traditional spreadsheet, ranges are used as an abbreviation for a set of neigbouring cells. For example, range *A1:C3* can be used to refer to 9 cells in columns A and C between rows 1 and 3. As such, a range represents a matrix of values and needs to be reduced to a scalar in order to be assigned to a cell as a value, which is usually done by applying a built-in function such as *sum()*.

Limited support for ranges in Lot would consist of parsing the range syntax and then allowing ranges to be used as arguments to functions such as *sum()* or *avg()*, as well as predicates like *distinct*. Another possible extension, quantifier predicates such as *all* (asserting that a relation should hold for all elements of a range) and *exists* (at least one element should satisfy the relation) would also support ranges. A possible application of such syntax is shown in fig. 5.1.

A more complete implementation would involve implementing matrix algebra to support arithmetic operations on both ranges and scalars. This would allow ranges to be freely used anywhere in the constraint expression. In this case, both sides of a constraint would need to reduce to a value of the same type (either two scalars or two matrices of same dimensions). To ensure that such constraint is valid, a simple type-checking would need to be performed.

5.4 Constraint replication

Cell replication is one of the most used features of a traditional spreadsheet. Whenever a user copy/pastes a cell, formula references from the original cell are automatically updated taking into account the relative locations of the dependent cells.

As constraints can be defined over multiple cells (for instance, *A1+A2+A3=s* from the magic square example), a different mechanism for replication needs to be developed that takes into account all constraints defined on the range of cells that are being replicated. In [Spe92], authors discuss one such approach that could be adopted for Lot.

5.5 Usability evaluation

Finally, an empirical evaluation in form of a usability study should be conducted in order to validate assumptions about the interaction model made in this thesis.

Conducting such study may however prove difficult at the current stage, as the prototype lacks many of the features that users presume when interacting with a spreadsheet, such as above discussed ranges, cell replication, dependency visualization and rich set of built-in functions.

CHAPTER 6

Related work

In this chapter, some of the early efforts at extending the spreadsheet with constraints are presented, as well as some alternative approaches such as bidirectional formulas and deductive spreadsheets. A more complete overview of the field can be found in [KV07] and [Cer13].

6.1 LogiCalc

Earliest attempts at extending the spreadsheet model involved combining spreadsheets with logic programming. First such spreadsheet was *LogiCalc* [Kri88], developed in 1983-88 by Frank Kriwazek as part of his Master thesis. *LogiCalc* was quite limited in functionality and could actually be considered more of a spreadsheet interface to define and query Prolog facts. Spreadsheet regions with values could be mapped to set of facts (for example, numbers in two columns could be mapped to a `days-in-month/2` relation). Users could declare constraints over defined relations using cells as variables to query the facts. For example, declaring:

```
father-of(A1,A2).  father-of(A3,A2).  father-of(A4,A3).
```

then supplying value for any of the *A1..A4* cells would supply values for the other cells that satisfied the constraints.

While constraints could also involve arithmetic predicates such as `Times(X,Y,Z)` (meaning $X * Y = Z$), such predicates weren't fully bidirectional as they required at least two instantiated variables (i.e. `Times(A1,A2,4)` could not find any solution).

6.2 PERPLEX

PERPLEX [Spe92] (fig. 6.1) supported working with constraints that were defined using Prolog-like predicates (infix notation was also supported, but internally rewritten to

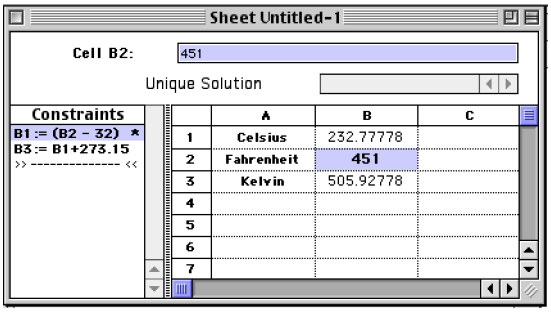

Figure 6.1: PERPLEX used to model the temperature converter example from chapter 3

(Source: Screenshot)

predicates). The built-in predicates had defined set of legal input-output modes for their arguments. For example, the `Plus(X Y Z)` predicate (meaning $X + Y = Z$) was defined using three functions:

$$\begin{aligned} f(X, Y) &= X + Y && \text{for mode (in in out)} \\ f(X, Z) &= Z - X && \text{for mode (in out in)} \\ f(Y, Z) &= Z - Y && \text{for mode (out in in)} \end{aligned}$$

This mode information was used to determine the order of evaluation of constraints, which also meant that a constraint `Plus(A1 A1 2)` could not be used to determine the solution $A1 = 1$. In such cases, another predicate called `Between` could be used to supply a range of possible values for an input, much like a for-loop in an imperative language.

PERPLEX also allowed users to define their own predicates in terms of the built-in ones, using *Programming by Example* [Hal84]: after a user defined relationship between cells in a spreadsheet using number of existing predicates, this relation could be saved as a new predicate, with automatically inferred input-output modes.

6.3 Equalizer

The constraints-based spreadsheet that Marc Stadelmann described in his Masters thesis [Sta93] was one of the first implementations that used an external solver. His prototype (called *Equalizer* as it supported only equality constraints) was implemented ontop of a commercial *Lotus Improv* spreadsheet and used the *Mathematica* software package for constraint solving.

Constraints are entered in a separate window outside the grid of cells, which was influenced by the separation between the data and computations already enforced in *Improv* [ABE07].

Improv pioneered a new addressing model of "named ranges" in which columns and rows of the spreadsheet grid can be hierarchically grouped and assigned names. Equalizer made use of named ranges by considering them as vectors and defining operations on vectors, allowing for more expressive constraint definitions.

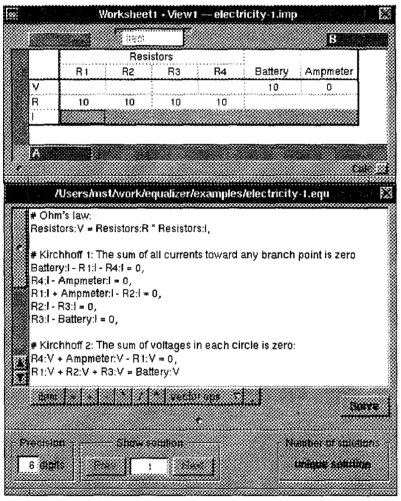

Figure 6.2: Equalizer used to model an electric circuit. Source: [Sta93]

6.4 Other systems

Extension of logic programming with constraints satisfaction, called constraints logic programming (CLP) [JL87], lead to number of implementations for specific classes of constraints, called CLP(X) where X stands for the constraint class (finite domains, arithmetic, linear etc.).

One such class, the CLP(fd) - with variables ranging over finite domains - has particular real-life applicability and has lead to number of spreadsheet extensions based on it, such as *ExSched* [CYG07], aimed at scheduling and timetabling problems and CsSolver [FFJ+03], developed with a particular use-case of on-site sales configuration and quotation in mind, both implemented on top of *Microsoft Excel*.

6.5 Bidirectional formulas

In [MPSC14], authors describe a framework based on concept of *lenses* [Fos10] - well-behaved bidirectional transformations - to synthesize reverse formulas from the ones input by the user. Implemented on top of *Microsoft Excel*, the add-in intercepts values entered by the user in cells holding a bidirectionalized formula, then applies the generated inverse formula to find the values for source cells and finally updates those cells instead. Authors claim that such approach is 1) intuitive, 2) not affecting the usual behavior of existing spreadsheets (conservative) and 3) presenting the new features using terminology that

	A	B	C	D	E	F	G	H	I	J
1	**students**	**courses**		**requires**		**hasTaken**		**skill**	**skillReq**	
2	Name	Initials	Weeks	Courses	Required	Student	Course	Name	Skill	Course
3	sue	an	3	an	bn	sue	bn	adv	adv	an
4	joe	bn	5	an	os	joe	os	base	adv	ss
5	ed	os	3	bn	ca	joe	ca		base	ca
6		ca	1			ed	ss			
7		ss	2							
8										

(a) Base relations

L	M	N	O	P	Q	R
coursesForSkill		**missingCourses**			**prepared**	
Skill	Course	Student	Course	Skill	Student	Skill
adv	an	sue	bn	adv	joe	base
adv	bn	sue	os	adv		
adv	os	sue	ca	adv		
adv	ca	sue	ca	base		
adv	ss	joe	an	adv		
base	ca	joe	bn	adv		
		joe	ss	adv		
		ed	an	adv		
		ed	bn	adv		
		ed	os	adv		
		ed	ca	adv		
		ed	ss	edv		
		ed	ca	base		

```
coursesForSkills(K,C) :-
  skillReq(K,C).
coursesForSkills(K,C) :-
  requires(B,C),
  coursesForSkill(K,B).
```

```
prepared(S,K) :-
  student(S),
  skill(K),
  not missingCourses(S,_,K).
```

```
missingCourses(S,C,K) :-
  student(S),
  coursesForSkill(K,C),
  not hasTaken(S,C).
```

(b) Deduced relations

Figure 6.3: Deducing relations in NEXCEL using Datalog rules. Source: [Cer13]

users are already familiar with (transparent) and see is as preferable to a constraints-based system.

6.6 Deductive spreadsheets

While some of the above systems use logic programming to implement the built-in functions and actual constraint solving, full power of logic programming - having the ability to specify and manipulate rules that can be used to deduce new facts - is not made available to the user.

[Cer13] calls spreadsheet extensions that allow Prolog-style computation of new values from existing values "deductive spreadsheets". The implementation described by Cervesato (fig. 6.3) uses Datalog, a syntactical subset of Prolog that has some properties beneficial in a spreadsheet-like environment, such as guaranteed termination of queries.

Other notable deductive spreadsheets include *XcelLog* [RRW07], also based on Datalog and supporting working with multiple values in each cell, as well as *PrediCalc* [KG07], which uses own variant of first order logic language and focuses on visualization and resolution of inconsistent constraints.

CHAPTER 7

Conclusion

A constraints-based spreadsheet, like the one implemented in this thesis, represents a fundamental change to a traditional spreadsheet model. Using constraints instead of unidirectional formulas not only increases the utility of spreadsheet as a tool for *what-if* analysis, but also increases its power as a *calculation tool*—making it applicable to a whole new set of problems.

Such fundamental change comes at a cost. Users need to adapt to the new model in which cell can hold both value and a formula, and become aware of implications that editing of cells has in the new model.

By utilizing a state-of-the-art Z3 SMT solver, we were able to profit from advances in SMT research over the past decade and rely solely on the solver for implementation of all arithmetics. Emscripten made it possible to compile the solver to JavaScript and run the whole application in the browser on the client-side. This let us avoid the need to perform roundtrips to the server for reevaluation, which might have introduced prohibitive latency for a highly interactive application.

The modern Web, defined by HTML5, CSS3 and emergence of new languages that compile to JavaScript, such as functional, statically typed language Elm used in this prototype, proved to be a highly productive development environment. Unparalleled ease of distribution (simply sharing an URL, with updated versions being available at the same URL) proved crucial in obtaining initial feedback on the prototype.

While it demonstrates advantages of a constraints-based model, our prototype lacks almost all other features found in a a modern spreadsheet. As implementation of these features would require multiple man-years of effort, the next step towards real-life application would be development of an extension for an existing spreadsheet such as Microsoft Excel—a challenge on its own.

Bibliography

[ABE07] Robin Abraham, Margaret Burnett, and Martin Erwig. *Spreadsheet Programming.* John Wiley & Sons, Inc., 2007.

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. *SIGPLAN Not.*, 48(6):411–422, June 2013.

[Cer13] Iliano Cervesato. *The Deductive Spreadsheet.* Cognitive Technologies. Springer, 2013.

[CYG07] Siddharth Chitnis, Madhu Yennamani, and Gopal Gupta. ExSched: Solving constraint satisfaction problems with the spreadsheet paradigm. *CoRR*, abs/cs/0701109, 2007.

[FFJ+03] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Christian Russ, and Markus Zanker. Developing constraint-based applications with spreadsheets. In Paul Wai Hing Chung, Chris J. Hinde, and Moonis Ali, editors, *Developments in Applied Artificial Intelligence, 16th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2003, Laughborough, UK, June 23-26, 2003, Proceedings*, volume 2718 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2003.

[Fos10] John Nathan Foster. *Bidirectional Programming Languages.* PhD thesis, University of Pennsylvania, 2010.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[Hal84] Daniel Conrad Halbert. *Programming by Example.* PhD thesis, University of California, Berkeley, 1984.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 111–119. ACM Press, 1987.

[KG07] Michael Kassoff and Michael R. Genesereth. PrediCalc: a logical spreadsheet management system. *Knowledge Eng. Review*, 22(3):281–295, 2007.

[Kri88] F. Kriwaczek. LogiCalc: A prolog spreadsheet. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, pages 193–208. Oxford University Press, Inc., New York, NY, USA, 1988.

[KV07] Michael Kassoff and André Valente. An introduction to logical spreadsheets. *Knowledge Eng. Review*, 22(3):213–219, 2007.

[Lel88] Wm Leler. *Constraint Programming Languages - Their Specification and Generation.* Addison-Wesley, 1988.

[MB09] Leonardo Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinícius Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 23–36. Springer-Verlag, Berlin, Heidelberg, 2009.

[Mic] Microsoft. Z3: Theorem prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[MPSC14] Nuno Macedo, Hugo Pacheco, Nuno Rocha Sousa, and Alcino Cunha. Bidirectional spreadsheet formulas. In Scott D. Fleming, Andrew Fish, and Christopher Scaffidi, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, pages 161–168. IEEE Computer Society, 2014.

[NON15] Fabio Niephaus, Philipp Otto, and Rzepka Norman. Emscripten-based solvers. Student project presentation for Constraint-Based Programming course at Hasso Plattner Institute. `https://github.com/normanrz/emscripten-constraints`, 2015.

[RRW07] C. R. Ramakrishnan, I. V. Ramakrishnan, and David Scott Warren. XcelLog: a deductive spreadsheet system. *Knowledge Eng. Review*, 22(3):269–279, 2007.

[Spe92] Michael Spenke. *PERPLEX - ein graphisch-interaktives System zur Unterstützung der logischen Programmierung.* PhD thesis, Universität Stuttgart, 1992.

[Sta93] Marc Stadelmann. A spreadsheet based on constraints. In Scott E. Hudson, Randy Pausch, Brad T. Vander Zanden, and James D. Foley, editors, *Proceedings of the Sixth ACM Symposium on User Interface Software and Technology, UIST 1993, Atlanta, GA, USA, November 3-5, 1993*, pages 217–224. ACM, 1993.

[Zak11] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.