The Design and Implementation of a Spreadsheet Based on Constraints

by

Marc Stadelmann
Licence en informatique de gestion, University of Geneva, 1988

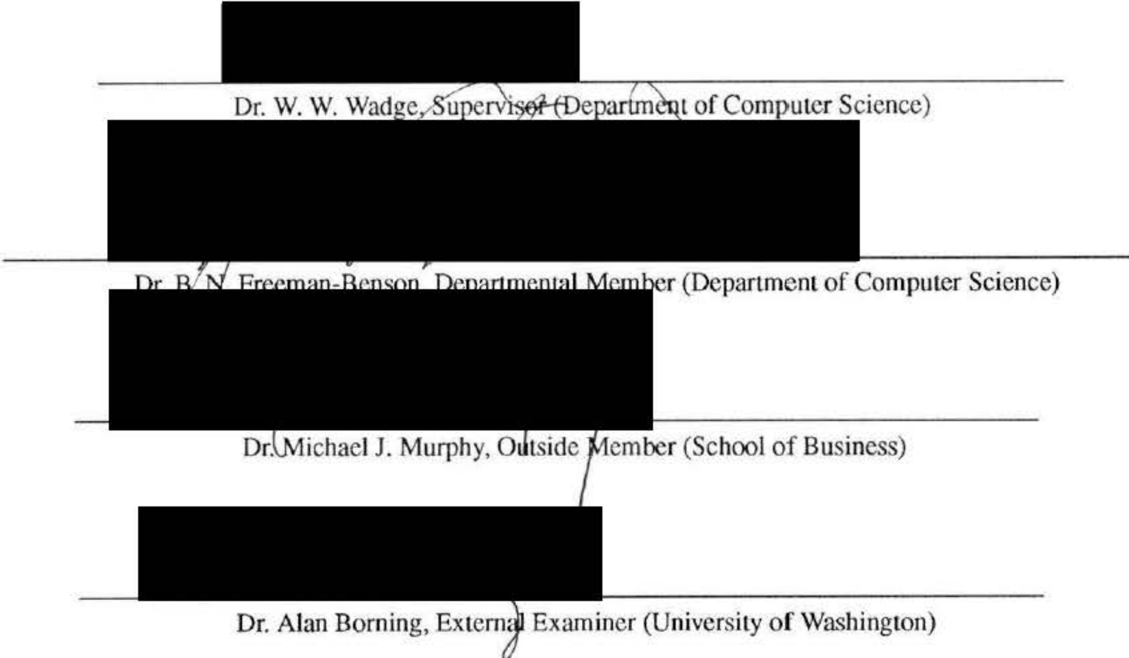
A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

ACCEPTED

MASTER OF SCIENCE

in the Department of Computer Science.

We accept this thesis as conforming
to the required standard



Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. B. N. Freeman-Benson, Departmental Member (Department of Computer Science)

Dr. Michael J. Murphy, Outside Member (School of Business)

Dr. Alan Borning, External Examiner (University of Washington)

© MARC STADELMANN, 1993

University of Victoria

All rights reserved. Thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Q340
S8

Supervisor: Dr. W. W. Wadge

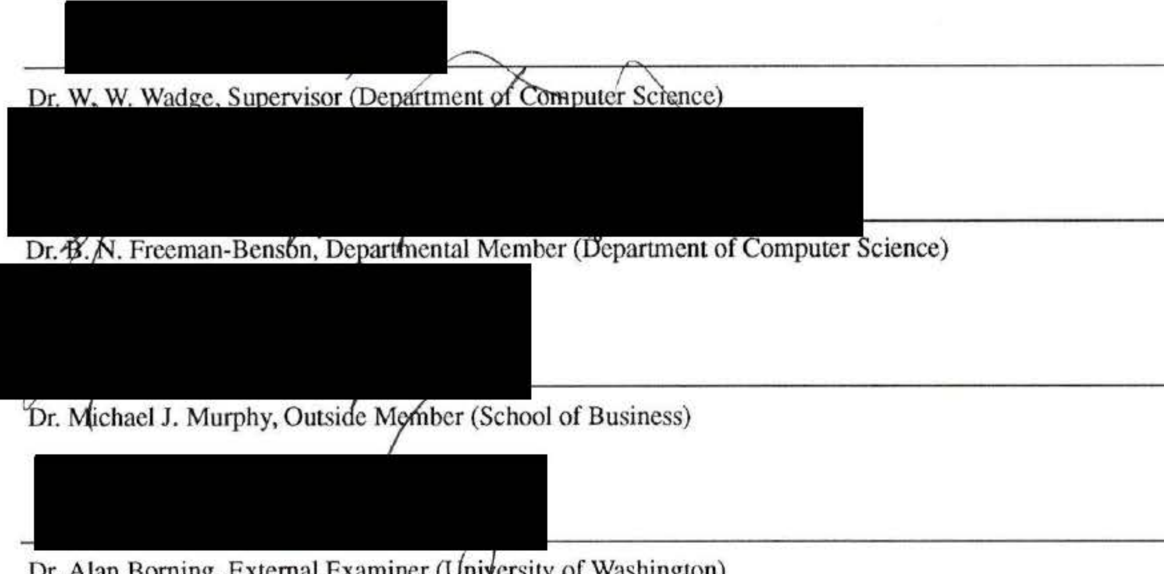
ABSTRACT

Constraints allow the user to declare relationships among objects and let the system maintain and satisfy these relationships. This thesis is concerned with incorporating constraints into spreadsheets. Instead of formulas, we let the user enter numerical constraints over the real values in the cells of the spreadsheet. Recalculating the spreadsheet then means (1) checking whether the given cell-values satisfy the constraints and (2) finding values for empty cells that satisfy the constraints.

We provide a brief overview of the typical conventional spreadsheet and give a formal (denotational) definition of its meaning (semantics). We then describe the design of our new spreadsheet and compare its semantics with the conventional spreadsheet's. The concept of constraints over cells is extended to constraints over arrays of cells (vectors). We introduce a language for constraints over the vectors of a two-dimensional spreadsheet and define several operations on these vectors.

We have implemented a subset of this new spreadsheet restricted to equality-constraints. A brief overview of this effort is provided and several implementation-related issues are discussed. We conclude this thesis by giving a collection of examples for which the idea of a constraint-based spreadsheet is well, or even uniquely, suited and we compare the solution of these examples with what conventional spreadsheets offer to solve the same problems.

Examiners:



Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. B. N. Freeman-Benson, Departmental Member (Department of Computer Science)

Dr. Michael J. Murphy, Outside Member (School of Business)

Dr. Alan Borning, External Examiner (University of Washington)

TABLE OF CONTENTS

List of Figures vi
Acknowledgements viii
Frontispiece ix

I Introduction

I.1 Motivation 1
I.2 Related work 5
I.3 Outline of this thesis 7

II Conventional Spreadsheets

II.1 Overview 8
II.2 Semantics of conventional spreadsheets 11
Syntax of formulas
Definitions
Semantic equations
Intuition
II.3 Circular dependencies and iteration 16

III A Spreadsheet Based on Constraints

III.1 Overview 19
III.2 Semantics of the constraint-based spreadsheet 22
Syntax of constraints
Definitions
Semantic equations
Comparison
III.3 Vector constraints and operations on vectors 26
Naming vectors
Adding, multiplying, etc. two vectors
Extracting cells and subvectors of a vector
Adding, multiplying, etc. all elements of a vector

III.4 Multiple solutions 29
Invertibility of functions
Answer constraints
Optimization

IV Implementation
IV.1 Overview 33
IV.2 Syntax and compatibility with Improv 36
IV.3 Solving power and limitations 37
IV.4 Precision of calculations 40
IV.5 Performance 41

V Application and Comparison
V.1 Simultaneous linear equations 45
V.2 Vector constraints 48
V.3 Goal seeking 48
V.4 Under-constrained problems and answer constraints 51
V.5 Multiple solutions 52

VI Conclusion and Future Work
VI.1 Contributions 53
VI.2 Future work 54

References 55

A Syntax 57

LIST OF FIGURES

Figure 1 Convert Fahrenheit into Celsius and vice-versa 3

Figure 2 Calculate the degrees where both Celsius and Fahrenheit are equal 4

Figure 3 A set of formulas with their corresponding dependency graph 10

Figure 4 The flat domain of the real numbers 13

Figure 5 Convergence and divergence of iteration in conventional spreadsheets 17

Figure 6 Naming of vectors 27

Figure 7 Operations on vectors 28

Figure 8 Optimization under constraints 30

Figure 9 Overview of the implementation architecture 34

Figure 10 Zero contour lines of two nonlinear equations f and g in two unknowns 37

Figure 11 Multiple solution browser and calculation precision controls 38

Figure 12 Example of a constraint-network 41

Figure 13 The dividend problem solved with the constraint-based spreadsheet 44

Figure 14 Example of an electrical circuit 44

Figure 15 A spreadsheet modelling an electrical circuit 45

Figure 16 Steady-state two-dimensional heat flow 47

Figure 17 Depreciation of an investment according to the year's digit method 48

Figure 18 Payments at the end of each year to pay back a ten year loan at a given interest rate 49

Figure 19 Multiple solutions 50

First of all I would like to thank Nigel for his off-hand help in arranging my stay here in Victoria. I remember his words: "It's raining again. Are you sure you want to come?". At the very same time, I would like to thank Bill, my supervisor, for his warm "Welcome on board" and for letting me so much freedom in choosing a topic and doing the work that I liked and the way I liked to do it. I can hardly imagine any better supervisor. Then, of course, I would like to thank Bjorn for the inspiration of my thesis topic and for his numerous suggestions and thoughtful comments and also for lending me a good portion of his library.

Then I would like to thank Kent, my best friend and Tennis partner here in Victoria, for his numerous distractive "Let's roll" and all these sad moments that we spent together... And last not least I would like to thank Victoria for providing all these opportunities to let wounds heal and get a fresh outlook on life and its delicacies.

And, last not least, I would like to thank my parents for their continuous support and encouragement.

I Introduction

I.1 Motivation

Electronic spreadsheets are widely-known, user-friendly calculation tools. Their commercial success is, to a large extent, based on their simple calculation model and their interactive interface. While spreadsheets' user-interfaces became more and more friendly, their calculation model has hardly evolved since they were invented little more than ten years ago. Spreadsheet calculation is based on assignment: the result of every formula is assigned to the cell where that formula resides. This simple model works well for the vast majority of conventional spreadsheet applications, namely business applications. However, as problems get more arduous, a spreadsheet's simplicity also becomes its limitation.

In this thesis, we propose a new spreadsheet based on the notion of a *constraint*. In one word, we replace formulas with constraints. A constraint, in general, allows the user to

express a relationship among objects and lets the system worry about maintaining and satisfying this relationship. In the case of a spreadsheet, we let the user declare numerical constraints between the real values of cells. Examples of constraints include:

- let two cells always be equal,
- let a cell be greater or equal to zero,
- let one cell be the sum of several other cells
- let one cell be twice as big as its neighbor
- etc.

Recalculating the spreadsheet then means (1) checking whether the given values in cells satisfy the constraints over these cells and (2) finding all solutions for empty cells that satisfy the constraints. In other words: given a set of constraints over a set of cells, and given values for some of these cells, the spreadsheet is able to mathematically derive all solutions to all remaining cells with respect to the given constraints. This process is referred to as *constraint satisfaction* or *constraint solving*. In the following, we call this new kind of spreadsheet the "new" or "constraint-based" spreadsheet as opposed to the "conventional spreadsheet."

This small shift in the spreadsheet paradigm has a rather significant impact on its problem-solving power compared to conventional spreadsheets. The most obvious improvement is that the flow of calculation is not hard-wired anymore into the spreadsheet's formulas. Instead, whichever cell is empty can be calculated according to the constraints that reign. Consider the following example where we declare a constraint between a cell named Fahrenheit:C1 and a cell named Celsius:C1[1]. Any value put into either of the two cells gets converted into the other temperature measurement.

Figure 1 Convert Fahrenheit into Celsius and vice-versa

	C1
Fahrenheit	
Celsius	32

1.8 * Celsius:C1 = Fahrenheit:C1 - 32

⇨

	C1
Fahrenheit	0
Celsius	32

1.8 * Celsius:C1 = Fahrenheit:C1 - 32

The same is not possible in a conventional spreadsheet: if we wanted to convert Celsius into Fahrenheit, we would have to write the formula **1.8 * Celsius + 32** into the Fahrenheit-cell and if, on the other hand, we wanted to convert Fahrenheit into Celsius we would have to algebraically transform the formula ourselves into **5/9 * (Fahrenheit - 32)** and write it into the Celsius-cell.[1] A constraint, on the other hand, acts like a true equation that can be solved for any unknown cell.

A second improvement is that a cell can be constrained by several constraints simultaneously instead of being calculated by one and only one formula. We can, for example, extend the above spreadsheet with a second constraint that requires both the Celsius-cell and the Fahrenheit-cell to be equal:

1. We reproduce the most widely used notation for referring to cells. Default names for rows and columns start with an 'R' or a 'C' respectively and are numbered from top to bottom and left to right respectively. Any row or column can be given a user-defined name. A cell is referred to by its row and column name, separated by a colon.

1. Note also that we could not have the formula calculating Fahrenheit in the Fahrenheit-cell and at the same time have the formula calculating Celsius in the Celsius-cell. This would introduce a circular dependency between the two cells.

Figure 2 Calculate the degrees where both Celsius and Fahrenheit are equal

	C1
Fahrenheit	
Celsius	

1.8 * Celsius:C1 = Fahrenheit:C1 - 32
Celsius:C1 = Fahrenheit:C1

⇨

	C1
Fahrenheit	-40
Celsius	-40

1.8 * Celsius:C1 = Fahrenheit:C1 - 32
Celsius:C1 = Fahrenheit:C1

This problem is in fact a set of two linear equations with two unknowns. Conventional spreadsheets only offer *iteration* to solve such kinds of problems. Iteration repeatedly calculates all formulas with the current cell values and eventually converges to a single result. However, the user has to first recognize the nature of his problem, then turn on the iteration feature and specify a condition for termination and a maximum number of iterations. However, even when the problem is known to have a solution, there is no guarantee that iteration finds it. For solving simultaneous equations reliably, one would have to resort to programming a specific algorithm in a spreadsheet's macro language.

A third improvement over conventional spreadsheets is that we always get *all* solutions to a particular problem. We can go even further and compute symbolic answers: if a problem has infinitely many solutions, we can get a more "simple" constraint as an answer that captures the relationship between the under-constrained cells. Or, alternatively, we can compute one "best" solution out of infinitely many by maximizing the value of a particular cell under the given constraints.

The goals of this thesis are to:

1. describe a design of such a new spreadsheet,
2. provide a formal definition of its meaning (semantics),
3. implement it and
4. give evidence of its practical potential.

There indubitably will remain unsolved problems. The bottom line argument of this thesis, however, is that (1) constraints drastically increase the problem-solving power of conventional spreadsheets and that (2) constraints can be quite seamlessly integrated into existing conventional spreadsheets.

I.2 Related work

We are not the first toying with the idea of a spreadsheet based on constraints. In the constraint-programming research community, a spreadsheet is often used as a vehicle to introduce and illustrate the basic concept of constraint programming. However, to the author's knowledge, no one from either research or industry has ever explored the idea to its full extent or published about it or implemented anything. As a matter of fact, beyond the vast number of very general books describing tips, tricks and traps of any commercial spreadsheet, there is very little published about spreadsheets. Among these publications, the following work is closely enough related to ours and should be mentioned and given credit.

Du [1] has designed and implemented a new kind of spreadsheet based on intensional logic. The motivation of Du's spreadsheet and ours is the same: extending the problem-solving power of conventional spreadsheets. The background and direction, however, are

very distinct. Both spreadsheets replace imperative formulas of conventional spreadsheets with a declarative language. Du's spreadsheet is based on a particular version of the dataflow language Lucid, pLucid, while ours is based on the notion of a constraint. The most characteristic difference is that pLucid is a full programming language and (as the name dataflow suggests) has a fixed direction in all calculations. Constraints, on the other hand, try to overcome exactly this limitation.

Spenke and Beilken [2][3] describe a spreadsheet interface for logic programming and mention the integration of constraints into spreadsheets. Their system, called PERPLEX, is based on an earlier idea of Van Emden et al. [4] and Krivaszek [5] who propose the raster layout of spreadsheets as a flexible user-interface to an interactive programming tool based on Prolog. Spreadsheet cells can be seen as Prolog variables and can be related through Prolog predicates. Prolog variables can (besides many other things) hold real numbers while predicates are like constraints. However, the emphasis of PERPLEX was not to be a more powerful *calculation tool* but, on the contrary, a more general *programming tool* whose applications go beyond numerical problems. Therefore, the solving capabilities for numerical constraints are fairly limited and "neither symbolic transformations of constraints nor iterative approximation algorithms are currently implemented" ([3] page 10).

Recently, some of the logic programming languages, such as Prolog III [7] and CLP($\Re$) [6], include constraints over limited domains and extend logic programming into constraint logic programming. To our knowledge, no one has worked on connecting a spreadsheet as an interface to any of these languages.

In a few words, all of the above proposals take an existing programming language and use a spreadsheet as its user-interface. We, on the other hand, introduce a shift in the way we look at a spreadsheet and obtain a more general and more powerful calculation tool.

I.3 Outline of this thesis

The next chapter provides and overview of the typical conventional spreadsheet and formally defines its semantics. This constitutes the background for designing our new spreadsheet in chapter III and lays the basis for a formal comparison between the conventional and our new constraint-based spreadsheet. We then extend the notion of a constraint over cells to constraints over vectors of cells and define some basic operations on such vectors.

We have implemented a subset of the new spreadsheet. Chapter IV shortly describes this effort and talks about several implementation-related issues, such as the solving power and limitations. The following chapter, Chapter V, gives examples from various application domains for which our new spreadsheet is better, or even uniquely, suited. Finally, we draw some conclusions in the last chapter and outline directions of future work.

II Conventional Spreadsheets

Before defining a new spreadsheet we want to have a good understanding of what a conventional spreadsheet is. It hardly makes sense to refer to specific products on the market; there are just too many of them. Instead, we shortly introduce and describe, in general, the typical conventional spreadsheet. We then use the formalism of denotational semantics to give a definition of a conventional spreadsheet's meaning (semantics). This will allow us to compare, on a formal basis, the typical conventional spreadsheet with the semantics of our new spreadsheet in the next chapter.

II.1 Overview

Briefly, a spreadsheet can be regarded as a pocket calculator, but with a friendlier, 2-dimensional user-interface. Spreadsheets can do basic arithmetic, such as addition, multiplication and exponentiation and they commonly provide a rich set of predefined functions for various problem domains, such as business applications and statistics. The

user-interface is organized into a grid of cells. The number of rows and columns of the grid can be adapted to fit the needs of a specific application. Cells can be addressed in various ways. Some spreadsheets label rows with letters and columns with numbers (or vice-versa) and a cell is referred to by a pair of letter and number. Other spreadsheets simply number rows and columns and refer to a particular cells with an expression like **R<rowNumber>C<columnNumber>**. Yet other spreadsheets allow the user to give labels to rows and columns and designate a cell with an expression **<rowName>:<columnName>** or any combination of the above.

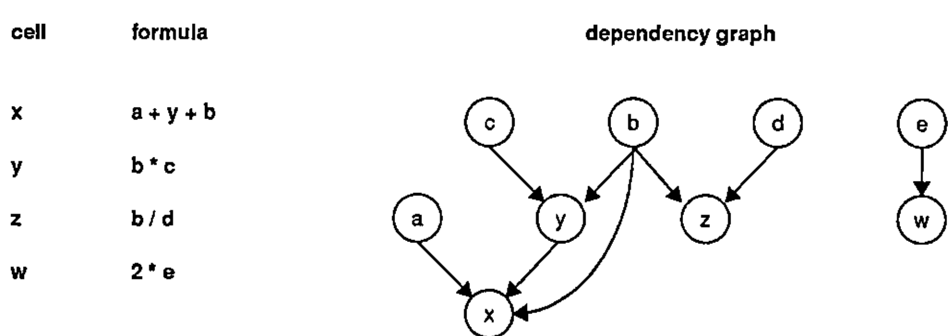
A cell can hold either text, numerical data, or a formula. Text in a cell is mainly used to document or label an adjacent cell, row or a column. Text-cells cannot be used in calculations[1]. The crucial point in a spreadsheet is that cells can be calculated by a formula that resides in that cell. Formulas most often contain references to other cells, which, in turn, may be calculated by a formula, introducing dependencies between cells. Every time the user enters a value or a formula into a cell, all dependent cells get automatically recalculated and updated as well. This is called *automatic recalculation* and can be seen as the most popular characteristic of spreadsheets, making them highly interactive.

The dependencies among a spreadsheet's cells can be visualized as a directed graph whose nodes represent cells and whose arrows represent the dependencies between these cells. An arrow arriving at a node means that this cell depends on the cell where the arrow starts. All source nodes, therefore, are cells that (are expected to) contain a value input by the

1. We ignore here that some spreadsheets have predefined operations on strings.

user. All other nodes are cells containing a formula. The sink nodes are formula-cells with no other cell depending on them. Note that the sum of all dependencies in a particular spreadsheet may form several, disconnected graphs. **Figure 3** illustrates all this with an example.

Figure 3 A set of formulas with their corresponding dependency graph



cell	formula
x	a + y + b
y	b * c
z	b / d
w	2 * e

The evaluation algorithm is straightforward. It has two phases:

1. If a formula is added or changed: topologically sort all cells referenced by constructing the dependency graphs as outlined above[1], then recalculate the value of the changed cell and continue with **2.**
2. Recalculate all cells "below" the cell that has changed (that depend on it) in an order that respects the dependencies specified in the dependency graph.

It is easy to see that as long as the graph does not contain a cycle (i.e. a path that, following the arrows, leads back to the start node) the recalculation procedure always terminates and delivers a unique result. A circular dependency, however, introduces a problem. We will

1. Note that in a smart implementation the graph does not have to be rebuilt from scratch but can be incrementally modified to reflect the change.

come back to this issue later in this chapter once we will have fully introduced and formalized all of conventional spreadsheets.

Most spreadsheets distinguish between absolute and relative cell references. The origin of this distinction helps understanding the consequence quite easily. Often, the same formula can be re-used to calculate not only one single cell but several other related cells in the same manner. The user can save a lot of work if he/she can write just one formula and then copy it into all cells without having to modify each individual copy in each destination cell. Relative references allow for exactly this. Whatever cell the formula gets copied into, a relative reference in that formula will be interpreted relative to the destination cell. It should be made clear that once all formulas are in place, all relative references can be translated into absolute references without any loss. As a matter of fact, relative references are simply a convenience for the user and do not add to the calculation power of a spreadsheet.

II.2 Semantics of conventional spreadsheets

In a similar way a language's syntax defines its form and structure, a language's semantics defines its *meaning*, that is, how it must be understood by the programmer or user. Within this thesis, we make use of semantics to (1) formally define what a conventional spreadsheet is and to (2) be able to compare the semantics of conventional spreadsheets with the semantics of our new constraint-based spreadsheet in the next chapter. To our knowledge, nobody has ever published any semantics of conventional spreadsheets. In the following, we will first define a simplified syntax for formulas in conventional

spreadsheets and then put together its semantics step by step. Semantic concepts are introduced as needed.

II.2.1 Syntax of formulas

Let's define a minimal spreadsheet with only the basic arithmetic operations. Formulas are simply expressions of the form[1]:

```
Expression    ::=    Expression '+' Expression
                   | Expression '-' Expression
                   | Expression '/' Expression
                   | Expression '*' Expression
                   | Value
Value         ::=    cellRef
                   | real
```

Note first, that it is unimportant for our purpose how, syntactically, cells are identified. Second, we make the assumption that all relative cell-references have been translated into absolute references.

A spreadsheet is more than just formulas: it matters where these formulas reside. Therefore, we regard a spreadsheet as an association from cells to formulas. We could express this association as a sequence of assignment statements:

```
Program       ::=    Assigment*
Assignment    ::=    cellRef ':=' Expression
```

1. Terminal symbols are italicized. A star denotes zero or more entities.

Notice, however, that although this syntax definition suggests sequentiality it does not require it. As a matter of fact, as we have seen it is not the syntax that defines the sequence of execution but the dependencies between the calculated cells.

II.2.2 Definitions

The denotational approach to semantics defines functions that map individual syntactical phrases to their intended meanings. Then, by composition, "the meaning of every composite phrase is expressed as a function of the meanings of its immediate sub-phrases" [8]. We will not provide a formal introduction into denotational semantics here but give enough intuition that is needed for our purpose. For a complete introduction we refer to [8].

We first introduce the notion of a *sheet*. A sheet is defined as an association of the finite set of cells C to the flat *domain*[1] of the real numbers, denoted $R_\perp$:

$$S = C \rightarrow R_\perp$$

In the following, we introduce the semantic functions and provide their intuition and domain of interpretation. The notation for semantic functions is somewhat different from normal mathematical functions. Semantic functions do not have explicit names but are

1. Intuitively, the semantic notion of a (computational) domain is explained as follows. In every computation there is a chance of non-termination. Our mathematical model of this computation has to take into account this possibility and include a result value of "I don't know (yet)", generally denoted as $\perp$("bottom"). A domain (1) defines a partial order that models approximation of computable entities. Starting with $\perp$, which means "no knowledge", we can successively improve a partial result with more computation. In addition, a domain (2) is defined to have an upper bound for every possible sequence (chain) of successive improvements.
The domain of real numbers is called flat because of its trivial structure, illustrated in **Figure 4**. It has $\perp_R$ at the bottom which may be improved to any of the real numbers eventually, after more computation.

defined by the syntactical phrase-type (within double square brackets) whose meaning they define. For example, [[X]] can be thought of as function f_X that defines the meaning of syntactical phrase types referred to by the syntactic meta-variable X.

Cells

Let C be the finite set of all cells. The meaning of a cell is a function that returns the value of that cell:

$$[[C]] = R_\perp$$

Formulas

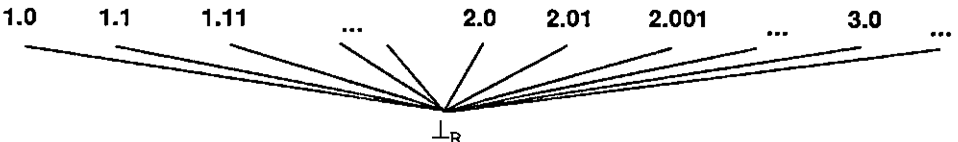
Formulas are expressions. Let E be the set of all formulas. The meaning of a formula is a function that, given a sheet of values, returns a real number, which is the result of that formula:

$$[[E]] = S \rightarrow R_\perp$$

Spreadsheet-programs

Let us define a *spreadsheet-program* (shortly *program)* as an association from the finite set of all cells to expressions (formulas). Notice that the program consists of all the formulas as well as all the values the user types into cells. Since no cell can possibly contain a

Figure 4 The flat domain of the real numbers



formula and a value input from the user at the same time, we can represent these input values as constant formulas for their cells. Therefore, the program completely represents a spreadsheet. Let P be the set of all possible spreadsheet-programs, let E be the set of all possible formulas and let C be the finite set of all cells, then programs are an association from cells to expressions like:

$$P = C \rightarrow E$$

The meaning of a program is the final sheet of values calculated by that program. Therefore:

$$[[P]] = S$$

II.2.3 Semantic equations

Finally, the semantic equations use the above functions to give a meaning to each syntactical phrase. Let `cellRef` (from the above grammar) be a reference to a cell `c` and let σ be a sheet. The meaning of a cell-reference, given a particular sheet, is the value of that cell in that sheet:

$$[[\texttt{cellRef}]]\sigma = \sigma(c)$$

Let E_1 and E_2 denote expressions (formulas) as defined in the above syntax. The semantic equations for expressions are straightforward:

$$[[E_1 + E_2]]\sigma = [[E_1]]\sigma + [[E_2]]\sigma$$

$$[[E_1 - E_2]]\sigma = [[E_1]]\sigma - [[E_2]]\sigma$$

$$[[E_1 * E_2]]\sigma = [[E_1]]\sigma * [[E_2]]\sigma$$

$$[[E_1 / E_2]]\sigma = [[E_1]]\sigma / [[E_2]]\sigma$$

We could easily have more operations and more complicated expressions with more semantic equations for these operations. However, the crucial equation that defines the meaning of the spreadsheet, would stay the same. The meaning of a spreadsheet-program π is defined recursively as:

$$[[\pi]]c = [[\pi(c)]]\ [[\pi]] \qquad \forall c \in C$$

II.2.4 Intuition

The intuition of the last equation is quite simple to understand. If we keep in mind that $[[\pi]]$ is the final sheet of values, i.e. the result of the program, then the equation can be understood as "the value of the sheet in cell c (left-hand-side) is the meaning (result) of the formula of that cell, evaluated in the final sheet (right-hand-side)". Alternatively, we could let σ be the result sheet and have the meaning of the program π defined as:

$$[[\pi]] = \sigma$$
$$\text{where } \sigma(c) = [[\pi(c)]]\sigma \qquad \forall c \in C$$

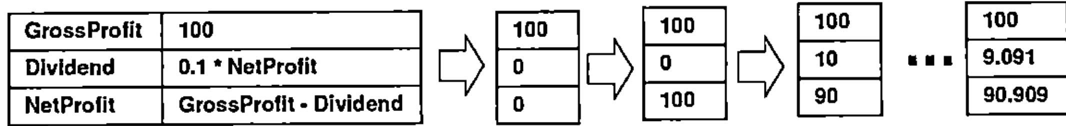
This is the *least* fixed point or, if there are no circular dependencies, the *only* fixed point.

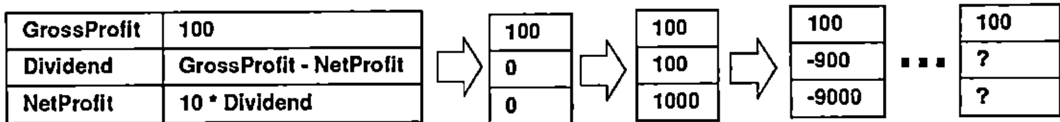
This concludes our formalization. In the next section we show that a spreadsheet's operational semantics (what a spreadsheet actually does) differs from its denotational meaning as defined in this section.

II.3 Circular dependencies and iteration

As we have seen in the first section of this chapter the evaluation algorithm always terminates as long as there are no circular references. In the presence of a circular

Figure 5 Convergence and divergence of iteration in conventional spreadsheets





reference, however, iteration does not reliably find the fixed-point if one exists. Consider the following example. A company decides to pay 10% dividend of its net profit. The amount of dividend paid in turn influences the company's net profit. The top left spreadsheet in **Figure 5** calculates the net profit and the dividend given the company's gross profit.

Obviously, this example contains a circular dependency. **NetProfit** depends on **Dividend** which in turn depends on **NetProfit**. First note that there is no other formulation that could possibly avoid the circle. In fact, the definition of the problem itself is circular. Most spreadsheets detect the circle and warn the user about the problem. Then the user has to (1) turn on iteration explicitly and (2) specify an end-condition together with (3) a maximum number of iterations the spreadsheet should perform. At each iteration, every formula gets evaluated once. If the formula contains references to an other formula-cell,

the current value — not the result of the formula — of that cell is taken, because evaluation of that formula could lead to an infinite loop. (Again, if the cell referred to is empty, zero is taken as a start value for the first iteration.) The top row of **Figure 5** gives the sequence of values calculated for our example problem. In this example, the sequence converges towards the correct solution, but unfortunately this is not always the case. The second spreadsheet in **Figure 5** is a slightly different but equivalent formulation of the same problem. However, no solution can be found this time because the consecutive values diverge and iteration stops after the maximum number specified by the user.

This is a rather vexatious shortcoming because the user must know under which condition iteration converges and formulate his problem accordingly. At a closer look, iteration turns out to be the Gauss-Seidel numerical method for solving sets of simultaneous equations. Our example problem, slightly transformed, consists of the following two equations:

$$Dividend = 0.1 \times NetProfit + 0 \qquad y = a_1 x + b_1$$
$$NetProfit = -1 \times Dividend + 100 \qquad x = a_2 y + b_2$$

In the general case, the Gauss-Seidel method for two equations converges under the condition $|a_1 \times a_2| < 1$, else it diverges. (If the product is equal to 1, most probably the user misformulated his problem.) For more than two equations, the conditions for convergence get more involved [16].

This concludes our analysis of conventional spreadsheets and sets the stage for defining our new constraint-based spreadsheet. This new spreadsheet introduces a shift in the basic spreadsheet paradigm from formulas to constraints and turns out to be a superset of the conventional spreadsheet as defined in this chapter.

III A Spreadsheet Based on Constraints

We have already introduced the key ideas of a constraint-based spreadsheet in the motivation of this thesis in the first chapter. This chapter will put forward a more comprehensive introduction and formalization. We first give an overview of the differences between a conventional and a constraint-based spreadsheet. Then, in the same way we did for the conventional spreadsheet, we introduce the semantics of a minimal constraint-based spreadsheet and compare both semantics. Thereafter, we extend constraints over cells to constraints over vectors of cells. Vector constraints take advantage of the two-dimensional grid of the spreadsheet for formulating constraints and considerably alleviate constraint writing.

III.1 Overview

The new spreadsheet's user-interface is very similar to a conventional spreadsheet. We have a two-dimensional grid of cells which can hold text or real numbers, but no formulas.

Instead of formulas, that calculate the value of one and only one cell, we have constraints that constrain the values of one or more cells. Moreover, cells can (and in practice often will) have not just one but several constraints on them. For these two reasons, we chose not to write a constraint into a particular cell but to collect all constraints together in a separate window, outside the grid of cells. This is what we did in the two motivating examples in **Figure 1** on page 3 and in **Figure 2** on page 4. To make it easier for the user to see which constraint(s) affect(s) which cell(s), we can highlight the cells when the user selects a constraint and highlight the associated constraints when the user selects a cell.

Recalculating the constraint-based spreadsheet is not as straightforward as recalculating a conventional spreadsheet. There are choices involved. Consider the example where A, B and C are cells with values 2, 3 and 5 respectively, satisfying the constraint $A + B = C$. If the user changes any of these cells, the spreadsheet does not know which of the other cells it has to adapt in order to satisfy the constraint. There are a variety of possibilities how the given constraint can be maintained. Let us assume the user has changed the value in cell C to 6, the spreadsheet can either:

- add 1 to cell A or add 1 to B or
- add 1/2 to both or
- add the same percentage to A and B so that the total amount added is 1
- or ...

depending on the problem we are trying to solve. There are strategies of how the user can tell the spreadsheet which cell to change and how. One could, for example, "freeze" certain cells and make their value read-only, which means they can only be updated by the user, but not be changed by the constraint-solver in order to satisfy a constraint. Or as an

alternative, one could ask the user which cell to adapt in the satisfaction process. Or hierarchies of constraints [17] with different strengths could be used to satisfy constraints that are absolutely required first and then proceed with less important constraints.

Besides the problem of which cells should be adapted when recalculating (if there is a choice at all) there is the question whether we should automatically recalculate every time the user changes a value or a constraint or whether we should let the user explicitly request to recalculate. It turns out that both approaches have their advantages and disadvantages. Automatic recalculation is closer to the behavior of conventional spreadsheets but the user may have to step through many uninteresting intermediate results (which may be expensive to compute) before actually solving the problem he or she has in mind. With manual recalculation, on the other hand, the cost of recalculating is only occurred when desired. However, while setting up a problem one does not get any feedback from the spreadsheet until one explicitly recalculates[1].

Both approaches are equally valid and one may only be favored over the other depending on the application at hand. For our purpose we simply assume manual recalculation: the user explicitly recalculates the spreadsheet every time he or she wishes. Recalculation then works as follows: whichever cell is empty gets recalculated under the given constraints and given values that the user input into the other cells. This is the behavior we have actually implemented (see next chapter) and it turns out to work quite well.

1. The constraint programming community similarly distinguishes two models for constraint solving: the *refinement model* and the *perturbation model* [17]. Automatic recalculation is a prime example of a perturbation model. Every time the user "perturbs" a cell (changes its value or formula) the spreadsheet is automatically recalculated. The refinement model, on the other hand, corresponds closely to manual recalculation: the user step by step refines the universe of solutions to finally end up with the solutions that are the answer to his problem.

III.2 Semantics of the constraint-based spreadsheet

We follow the same steps for defining the semantics of constraint-base spreadsheet as we did in the previous chapter for the conventional spreadsheet.

III.2.1 Syntax of constraints

As the constraints are not written into cells we have a complete linear constraint language. While formulas can be interpreted as assignments, with a cell as the left-hand-side of the assignment operator and the right-hand-side being the formula (expression), constraints are more general relations with expressions to both sides of the relational operator. Let us define a minimal syntax for constraints:

```
Program      ::=    Constraint 'AND' Program
                  | Constraint 'OR' Program
Constraint   ::=    Expression '=' Expression
                  | Expression '>' Expression
                  | Expression '<' Expression
                  | Expression '>=' Expression
                  | Expression '<=' Expression
Expression   ::=    Expression '+' Expression
                  | Expression '-' Expression
                    Expression '*' Expression
                    Expression '/' Expression
                    Value
Value        ::=    cellRef
                  | real
```

III.2.2 Definitions

Cells

The function defining the meaning of a cell returns a real value. As a matter of fact, we don't need $\perp$ bottom anymore, because a cell either satisfies the constraints or not; there is no possible "don't know (yet)". Therefore, let C be the finite set of cells and let R be the real numbers. Then:

$$[[C]] = R$$

Constraints

Let S be the set of all possible sheets as defined in the previous chapter. Let Q be the set of constraints and let B be the set of booleans {true, false}. The function defining the meaning of a constraint takes as input a sheet and returns a boolean value, saying whether the constraint is/can be satisfied:

$$[[Q]] = S \rightarrow B$$

Constraint-based spreadsheet-programs

Again, we represent the values the user inputs into cells as constant equality constraints that fix the value of these cells. We redefine a spreadsheet-program as being a set of constraints (as opposed to "an association of cells to formulas"). A program is a subset of all possible sets of constraints:

$$P = \wp(Q)$$

The function defining the meaning of the program changes as well. Unlike a conventional spreadsheet, a constraint-based spreadsheet is non-deterministic and can have more than one solution. Each solution is represented as one sheet of values that satisfies the given

constraints. Therefore, the meaning of a constraint-based spreadsheet-program is a subset of sheets out of the set of all possible subsets of sheets:

$$[[P]] = \wp(S)$$

III.2.3 Semantic equations

The equations for cells and expressions do not change. Let `cellRef` be a reference to cell c and let σ be a sheet. Then:

$$[[\texttt{cellRef}]]\,\sigma = \sigma(c)$$

and let E_1 and E_2 be expressions:

$$[[E_1 + E_2]]\,\sigma = [[E_1]]\,\sigma + [[E_2]]\,\sigma$$
$$[[E_1 - E_2]]\,\sigma = [[E_1]]\,\sigma - [[E_2]]\,\sigma$$
$$[[E_1 * E_2]]\,\sigma = [[E_1]]\,\sigma * [[E_2]]\,\sigma$$
$$[[E_1 / E_2]]\,\sigma = [[E_1]]\,\sigma / [[E_2]]\,\sigma$$

Let C_1 and C_2 be constraints and let E_1 and E_2 be expressions. If σ is a sheet the semantic equations for the comparison operators become:

$$[[E_1 = E_2]]\,\sigma = \begin{cases} \textit{true} \text{ if } [[E_1]]\,\sigma = [[E_2]]\,\sigma \\ \textit{false} \text{ otherwise} \end{cases}$$

$$[[E_1 > E_2]]\,\sigma = \begin{cases} \textit{true} \text{ if } [[E_1]]\,\sigma > [[E_2]]\,\sigma \\ \textit{false} \text{ otherwise} \end{cases}$$

etc.

And:

$$[[C_1 \text{ AND } C_2]]\sigma = [[C_1]]\sigma \text{ and } [[C_2]]\sigma$$
$$[[C_1 \text{ OR } C_2]]\sigma = [[C_1]]\sigma \text{ or } [[C_2]]\sigma$$

Finally, let K be the set of constraints of program π and let κ be a constraint out of K. The meaning of the constraint spreadsheet-program π is:

$$[[\pi]] = \{ \sigma \mid [[\kappa]]\sigma = \text{true } \forall \kappa \in K \}$$

III.2.4 Comparison

Not surprisingly, the meaning of a constraint-based spreadsheet is quite different from a conventional spreadsheet. First of all, the solution sheet can not contain any cell with $\perp$ in it. Either the value of the cell satisfies the constraints on it or it doesn't; there is no incomplete answer. Second, the constraint-based spreadsheet is non-deterministic and gives all solutions to a particular problem while a conventional spreadsheet always gives one and only one answer. However, the solutions given by the constraint spreadsheet will always include the answer given by its corresponding conventional spreadsheet. Therefore, we can say that constraint-based spreadsheets are a meaning-preserving extension of conventional spreadsheets.

This concludes our formalization. In the rest of this chapter, we will introduce a number of features that make a constraint-based spreadsheet more user-friendly and problem-oriented.

III.3 Vector constraints and operations on vectors

Assembling constraints outside of the grid of cells coalesces all information in one single area. However, even a simple spreadsheet application can easily have more than a dozen constraints, probably about as many constraints a similar conventional spreadsheet would have formulas. Like formulas, many of these constraints calculate cells in the same manner. We introduce the concept of a constraint over a *vector* of cells that takes advantage of the spreadsheet's grid to write sets of constraints more compactly. We define a vector as two or more adjacent cells in one row or one column. There are several types of operations that we can perform on vectors. Let us take for example addition.

1. A single cell (scalar) can be added to (every cell of) a vector.
2. Two vectors can be added together. All operations on vectors are defined point-wise, that is, cell by cell. Therefore, the size of two vectors, i.e. the number of cells, must be the same.
3. We can extract cells or subvectors of a vector
4. We can sum all elements of a vector.

This leads to an APL-like constraint-language. Notice here that, like in APL, we could define corresponding operations on matrices as well. We will introduce and illustrate the above operations in the following.

III.3.1 Naming vectors

Let us first define a naming scheme for vectors. The simplest scheme would be to have as vectors only the entire row or column of cells and then use the name (or number) of this row or column. For most but the simplest applications such a scheme appears not to be flexible enough. We propose to group a number of adjacent rows or columns together into a *group* and give the group a separate name. A group, basically, allows to shorten an entire

Figure 6 Naming of vectors

	Resistors				Battery	Ampmeter
	R1	R2	R3	R4		
V						
R						
I						

row or column vector to a desired length and can be nested. In **Figure 6**, for example, Resistors groups together columns **R1** to **R4**.

III.3.2 Adding, multiplying, etc. two vectors

Figure 6 shows a spreadsheet that calculates current and resistance in a simple electric circuit. Ohm's law for all four resistors **R1** to **R4**, for example, can be expressed by one single vector constraint:

V:Resistors = R:Resistors * I:Resistors

III.3.3 Extracting cells and subvectors of a vector

We introduce four structural operations on vectors, **FIRST()**, **LAST()**, **ABF()** and **ABL()**. **FIRST(*vectorName*)** and **LAST(*vectorName*)** extract the first and last cell of a vector respectively. The first cell of a vector is its leftmost or topmost cell, depending on whether the vector is horizontal or vertical. The last cell is defined correspondingly[1].

ABF(*vectorName*) gives "all but the first" cell of a vector and **ABL(*vectorName*)** returns

"all but the last" cell of a vector. Together, these four operations allow to extract any subvector of a given vector. **Figure 7** gives an example of how these operations can be used. The example spreadsheet calculates the depreciation of an investment according to the year's digits method. This method is used when the depreciation is the greatest during the first few years of use. For example, suppose a drilling machine that initially costs $15,000 has a life span of five years and can be resold after for $5,000. The total depreciation is $10,000. The sum of the years 1 through 5 is 1+2+3+4+5=15. Thus, the depreciation for the first year is 5/15, for the second 4/15, etc. The digits for each year can be calculated as follows:

```
LAST(digits:years) = 1
ABL(digits:years) = ABF(digits:years) + 1
```

Notice that the two vectors on the left and right hand side of the equal sign in the second constraint overlap. This technique allows us to calculate sequences of values within a vector. This would normally be achieved as many formulas with relative references as there are cells in that vector.

III.3.4 Adding, multiplying, etc. all elements of a vector

These operations are called reduction operations on vectors. Reduction is equivalent to placing a dyadic operator between each element of the vector and evaluating the resulting expression. For example, the sum of the year's digits is computed by:

```
SUM(years:digits) = digits:investment
```

1. Note here that we chose not to differentiate between a horizontal and vertical vector in the language (and for example refer to the first cell of a vertical vector with "top") because in most spreadsheets one can interchange rows and columns.

Figure 7 Operations on vectors

		years				
	investment	year1	year2	year3	year4	year5
value	$15,000					
digits	15	5	4	3	2	1
depreciation		$2,000				

Then we calculate the depreciation for each year. It is sometimes convenient to have a new variable that is not part of the sheet itself, like **dep**, for the total depreciation of the article:

dep = FIRST(value) - LAST(value)

ABF(depreciation) = dep * digits:years / digits:investment

As we have demonstrated with this example, vector constraints considerably reduce the amount of work involved in constraint writing. There are other advantages as well. First, we can bypass the need for relative references. As the scope of a constraint can be an array of cells and we do not need to write a formula with relative references and copy it to all the cells. Second, we can extend our spreadsheet and add rows and columns (for example to calculate depreciation for 6, 7 or 10 years) without changing a single constraint. And third, it becomes easier editing the constraints.

III.4 Multiple solutions

A set of constraints can have any number of solutions, from none to infinitely many. Besides the problem of computing these solutions (which we will discuss in the next

chapter on the implementation), we are concerned with how we handle and display them to the user. For most practical applications, the user is most probably interested in one particular solution or the "best" solution out of many. We will consider the most common cases here and outline solutions.

III.4.1 Invertibility of functions

Some functions, such as sin, cos, etc. are not uniquely invertible. In fact, there are infinitely many solutions to the equation $\sin(x) = 0.5$. For most applications, however, it makes sense to provide the primary inverse by adding the constraints $x \leq 90$ and $x \geq 0$ by default. Similarly, we can add the constraint $x \geq 0$ for an expression like $\sqrt{x}$ to filter out any negative solutions. Other functions, such as the modulo (rest of a division), have no "primary" inverse. The answer to a constraint like $35 \otimes x = 5$ (where the circle operator stands for the modulo) is $x = 10, 20, 30$, none of which is any better than the other. Yet other functions, such as the maximum, are invertible only in certain cases. For example, $6 = max(x, 3)$ is invertible while $6 = max(x, 9)$ is not.

III.4.2 Answer constraints

This leads us to the more general case when a problem is *under-constrained*, that is it has infinitely many solutions, none of which is any better than any of the others. Instead of arbitrarily choosing one solution over the other it is more helpful for the user to get back one or several more "simple" constraints that capture the relationship between the under-constrained cells. This means to symbolically transform and "simplify" the given constraints. Notice that the term "simplify" is subjective and depends on the problem at

Figure 8 Optimization under constraints

		product			total
		steel1	steel2	steel3	
resource	iron	40	45	25	20,500
	coal	50	25	50	28,000
	oil	10	30	15	11,500
quantity					

hand into which a symbolic solver has no insight. However, in general we can say that an answer constraint formulates a condition that every solution to the given input constraints must satisfy. Answer constraints can be seen as the result of a partial evaluation of the spreadsheet. We will give an example in chapter V.

III.4.3 Optimization

In most practical problems that have multiple solutions the user is interested in a "best" solution, which implies optimization. Optimization fits in perfectly with constraints: we optimize a function under a given set of constraints (inequalities). Consider the linear programming problem in **Figure 8.** We have three different qualities of steel. Each kind of steel takes needs the given proportions of raw material. Given a finite amount of raw material and the selling price for each kind of steel, we would like to maximize the profit. Let us assume **steel1**, **steel2** and **steel3** sell for $111, $222 and $333 respectively. If we

extended our constraint language with a command like **MAXIMIZE()** we could write something like:

```
MAXIMIZE( 111 * steel1:quantity + 222 * steel2:quantity + 333 * steel3:quantity)
```

under the constraints:

```
product:quantity >= 0
SUM( product:iron * product:quantity) <= total:iron
SUM( product:coal * product:quantity) <= total:caol
SUM( product:oil * product:quantity) <= total:oil
```

IV Implementation

We have implemented a subset of the new spreadsheet restricted to equality constraints (equations). The goal of this implementation is to give evidence of the practical potential of the idea. In this chapter, we first provide an overview of the implementation effort and then discuss several implementation-related issues in more depth.

IV.1 Overview

Equality constraints are the most useful type of constraints for spreadsheets and can be readily solved with standard mathematical packages. Inequalities, on the other hand, more often allow for infinitely many solutions. For the scope of this thesis, we restrict ourselves to implementing equality constraints or simply equations. Because of this restriction we call the resulting spreadsheet the "Equalizer". The resulting spreadsheet, however, is a superset of the capabilities of a conventional spreadsheet.

The Equalizer has been developed in Objective-C© on a NeXT© computer, NextStep 2.1. Overall, we were pleased with NeXT's development environment to an extent that our very few criticisms are largely outweighed by the vast benefits. Objective-C is an attractive and easy-to-learn language and appears to be much cleaner and less overloaded with features than its competitor C++. InterfaceBuilder©, NeXT's tool to interactively design an application's interface, saved us hours of tedious graphical programming. But it is not even the individual tools and their quality that impressed us most but their seamless integration. We use lex, yacc, inter-process-communication libraries, InterfaceBuilder and make them work together with two off-the-shelf applications without even writing a Makefile for the compilation.

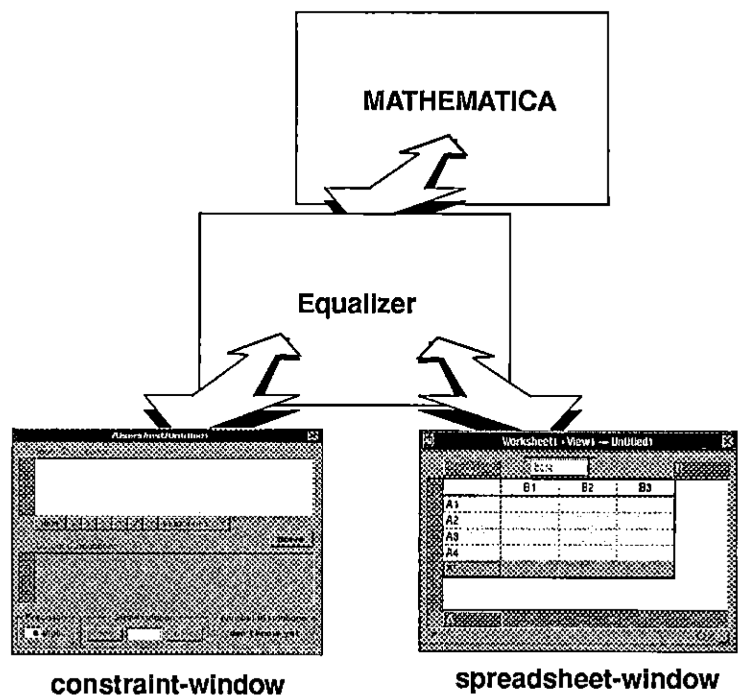
The implementation itself consists of three parts:

1. a window with a grid of cells called the *spreadsheet-window*,
2. a window with the constraints to solve, called the *constraint-window*, and
3. an internal *constraint solver* engine.

Instead of implementing a full spreadsheet interface with all its editing and manipulation capabilities we make use of the interface of an existing spreadsheet, Improv© from Lotus© as the spreadsheet-window. Improv comes along with an application programming interface, that gives the programmer access to cells and their values from within his application.

The second window, the constraint-window, is the interface of the Equalizer properly. It allows the user to enter and edit constraints and provides several other interface components such as buttons, a solution browser and a panel to display answer constraints. At the back-end of the Equalizer, we replaced the conventional spreadsheet calculation

re 9 Overview of the implementation architecture



engine with a constraint-solver. Again, we make use of an existing mathematics package, MATHEMATICA©, for solving constraints. Finally, the code we have written glues all the parts together. We parse the constraints in the constraint-window and retrieve the structure of the spreadsheet and the values of cells from the spreadsheet-window. We check the input for consistency and then translate it into a suitable form for MATHEMATICA to solve. MATHEMATICA returns either a numerical or a symbolic solution, which we analyze, translate and display either as a set of values in the spreadsheet-window or as a set of answer constraints in the constraint-window. The architecture described is

illustrated in **Figure 9**. It allows us to have a full-fledged implementation with a relatively modest implementation effort of about five thousand lines of source code.

We do not give any further details of the implementation here as they are of little interest for any reader but the programmer. Instead, we focus on several implementation-related issues: compatibility with Improv, precision of calculations and performance.

IV.2 Syntax and compatibility with Improv

The idea of collecting all formulas in one place instead of writing each of them into its respective cell is not ours. As a matter of fact, Improv is the first spreadsheet, to our knowledge, that implements this concept. Improv's formulas are written sequentially at the bottom of the spreadsheet. Syntactically, they are of the form:

```
Formula        ::=     CellRef '=' Expression
```

where the equal sign is interpreted as single assignment. The Equalizer's constraints extend Improv's formulas to allow for arbitrary complex expressions on both sides of the equal sign, which is now interpreted as numerical equality:

```
Equation       ::=     Expression '=' Expression
```

This makes the Equalizer syntactically downward compatible with Improv spreadsheets and every Improv spreadsheet can be solved by the Equalizer.

We have implemented all basic operations (+,-,*,/,^) on vectors, plus the four structural operations **FIRST()**, **LAST()**, **ABL()** and **ABF()**, but no reduction operators, such as the **SUM()**. All sizes of vectors are checked for compatibility with the operations performed

on them. The full syntax of the Equalizer is given in appendix A. To distinguish the values input by the user from the values calculated by the Equalizer we display the latter in bold font.

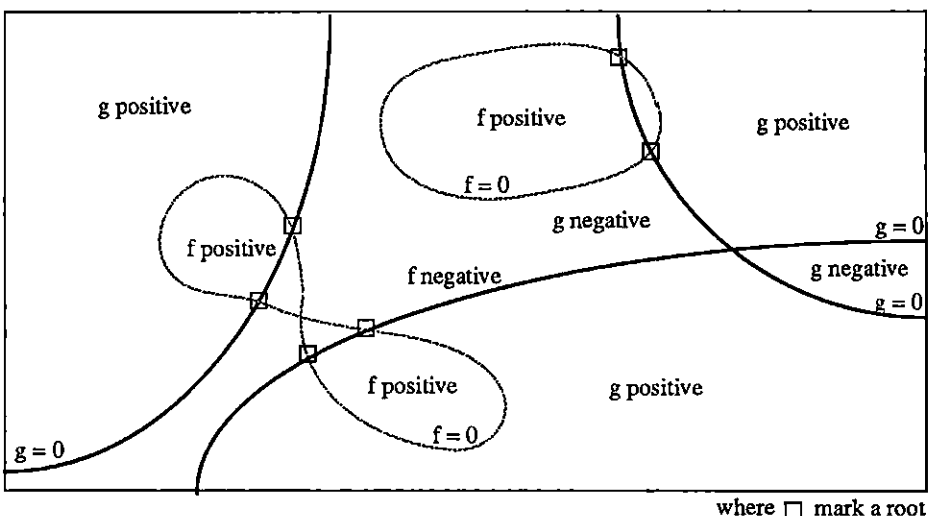
IV.3 Solving power and limitations

As already mentioned, we use MATHEMATICA, an existing mathematics package, for solving equations. MATHEMATICA can find numerical approximations to the solutions of any polynomial equation and any system of linear equations. Unfortunately, it is not documented what algorithm is used. We assume that a combination of symbolic manipulations and numerical techniques, such as Gaussian Elimination, are used.

In the case of more than one non-linear equation the solution gets more complicated. The authors of [13] put forward a very elegant argument why no good, general method can (and most likely never will) be found for systems of more than one non-linear equation:

> Consider the case of two dimensions, where we want to solve simultaneously:
>
> $$f(x, y) = 0$$
> $$g(x, y) = 0$$
>
> The functions f and g are two arbitrary functions, each of which has zero contour lines that divide the (x,y) plane into regions where their respective function is positive or negative. These zero contour boundaries are of interest to us. The solutions that we seek are those points (in any) which are common to the zero contours of f and g (see **[Figure 10]**). Unfortunately, the functions f and g have, in general, no relation to each other at all! There is nothing special about a common point from either f's point of view, or from g's. In order to find all common points, which are the solitons of our non-linear equations, we will (in general) have to do neither more nor less than map out the full zero contours of both functions. Note further that the zero contours will (in general) consist of an unknown number of disjoint closed curves. How can we ever hope to know when we have found all such disjoint pieces?
> For problems in more than two dimensions, we need to find points mutually common to N unrelated zero-contour hyperplanes, each of dimension $N-1$. You see that root finding becomes virtually impossible without insight!

Figure 10 Zero contour lines of two nonlinear equations f and g in two unknowns

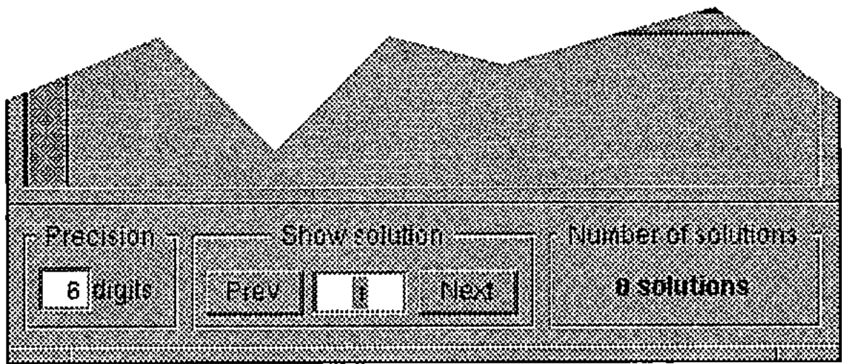


Finding numerical solutions to more general (systems of) equations such as equations involving transcendental functions or exponential equations is even more difficult. An extreme example (due to Borning [10]) consists of finding a solution to the equation:

$$x^n + y^n = z^n$$

for $x, y, z, n \in N$ and $x, y, z, n > 2$. Any such solution would be a counter-example to Fermat's last (unproven) theorem which states that no such solution exists. Symbolic answers are yet another problem to compute. For example, if the highest power in a polynomial equation is five or more it may be mathematically impossible to give explicit algebraic formulas for all solutions, [9] page 96. Moreover, some solutions to polynomials may not be real but complex. Fortunately, for many problems of practical interest, these limitations never become an important factor.

Figure 11 Multiple solution browser and calculation precision controls



If a problem has more than one but a known fixed number of solutions a small panel below the constraint-window displays the number of solutions found. The user can browse through them and view one after the other (see **Figure 10**). Other kinds of methods for browsing multiple solutions, such as [10], could easily be imagined.

As a conclusion, we can say that the problem-solving power of a constraint-based spreadsheet is limited to whatever can be solved in numerical and symbolic mathematics, whereas the limitation of a conventional spreadsheet is the solving-power of one particular algorithm, namely the Gauss-Seidel algorithm. This suggests a basic constraint-based spreadsheet with a quite general solver, that can be extended and specialized (possibly by the user himself) to solve problems of a particular application domain very efficiently.

IV.4 Precision of calculations

We all know that "squeezing infinitely many real numbers into a finite number of bits requires an approximate representation" [11]. For a conventional spreadsheet and its applications this loss of precision is of little concern. However, as we are dealing with true equations, tests for equality of two numbers are crucial. For instance, are the numbers 0.1 and 0.100000001 equal? The nasty consequence of this problem may be that we manage compute a solution that satisfies all constraints, but, as we solve again with the previous answer left as an input (asking whether this is indeed a solution), the answer may be no.

A simple but partial solution is to round all numbers to a certain number of digits after the comma. In many calculations, however, the precision of the result progressively degrades due to roundoff errors. As a consequence, the result has less significant digits than the initial operands. In our implementation we provide a field in which the user can specify a number of decimal digits each number is rounded to (see **Figure 10**). We can not control, however, the accumulation of round-off errors nor the precision with which equality tests are performed within MATHEMATICA.

Lee [14] proposes a solution to this problem in the case of the constraint logic programming, based on interval arithmetic. It consists of bracketing floating point numbers within an interval that is adjusted during computation. Two numbers whose intervals overlap are considered to be equal. Within the scope of this thesis, we confine ourselves to emphasizing that (1) problems associated with the precision of calculations do not compromise the process of finding solutions. However, testing whether a given

solution conforms to the constraints may be problematic. Therefore, we can only say that (2) comparisons between approximate real numbers must be treated with care.

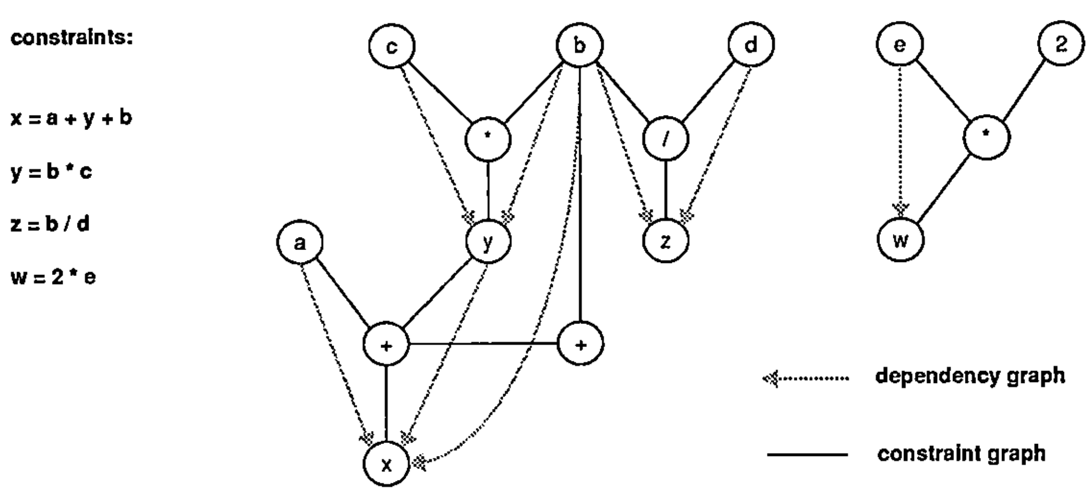
IV.5 Performance

The performance of our implementation is fairly low[1]. On one hand, a fixed part of the loss in performance is due to the architecture of the implementation. There are three rather large applications running at the same time, communicating with one another. As a consequence, the machine is swapping a lot, which makes the performance degrade considerably. A commercial, single process implementation can be much faster in that respect. On the other hand, the response time depends on the amount and numerical complexity of the calculations involved. In our implementation, even a straightforward, conventional spreadsheet (single assignment, without circular references) is solved as a set of simultaneous equations and consequently takes longer to solve in MATHEMATICA than with a conventional spreadsheet.

Ideally, one would like to pay the cost of a more powerful algorithm only when necessary and solve conventional applications equally efficiently as with a conventional spreadsheet. In the following, we outline an implementation based on *constraint-networks* that has similar performance for solving conventional spreadsheet applications with no circular references, which constitute the vast majority of existing spreadsheet applications.

1. To give an idea: a problem with approximately a hundred simultaneous liner equations is solved within fifteen seconds on a low-end NeXT computer.

Figure 12 Example of a constraint-network



Constraint networks are a solving technique commonly used in the constraint-programming (see for example [12]). Constraint networks are very similar to the dependency graphs which we introduced in chapter II. There are two important differences. First, a constraint-graph has additional nodes for every operator as well as every constant. Second, the arcs of a dependency graph are directed and indicate a dependency between the two nodes (variables), while the arcs of a constraint-network are not directed and simply express participation of a variable in an operation. **Figure 10** gives an example of a constraint-network. It takes the same example problem as **Figure 3** on page 10 to illustrate dependency graphs. This time, however, we interpret the formulas as constraints. The constraint-network is drawn in black, while the corresponding dependency graph is shown in slight gray.

Constraint-networks are evaluated by a *local propagation* algorithm. It propagates known values of variables through the net. For example, given values for the variables a, b, c, d and e, the values of x, y, z and w can be computed. In general, one can always deduce values for all variables in the network if:

1. every operator-node has enough input to compute an output and
2. at least one variable is known within every cycle.

It is easy to see that both conditions are always met by all conventional spreadsheet problems. First, if an operator node needs a value that cannot be deduced we can take its value to be zero, as conventional spreadsheets do (or we ask the user for its value). The second condition is satisfied by our hypothesis that we have no circular dependencies. A cycle in a dependency graph (a circular dependency) is similar to a cycle in the corresponding constraint-network, except that the latter cycle passes through additional operator nodes. If, in a constraint-network, none of the values within a cycle is known or can be deduced, it means that the corresponding dependency graph must have a circular dependency[1], because in a circular dependency all variables are defined, directly or indirectly, by themselves (and, therefore, none them can be deduced from the others). This, however, has been ruled out by our hypothesis. Thus all conventional applications can be solved with constraint-networks.

It is also easy to see that the cost of building and evaluating a constraint-network is within a constant factor of the corresponding conventional spreadsheet. The data-structures of a constraint-network are slightly bigger but grow at the same rate as their corresponding

1. This is not exactly true. There is in fact one case, where a constraint-network has a cycle that does not correspond to a circular dependency. Consider the network for $x \times x = y$. The two arcs from the x-node to the multiplication-node form a minuscule cycle. It turns out that these cases can easily be eliminated. See [15]

dependency graph. Therefore, the cost of building the constraint-network is almost the same[1]. Evaluation of a constraint-network does not involve more work than evaluating a conventional spreadsheet. Every propagation step is part of an evaluation of an expression (formula) and every deduction of a new value corresponds to assigning the result of the expression to a cell. Thus, the total additional cost (for problems with no circular dependencies) is only a constant factor more than for conventional spreadsheets. The gain, however, is that constraint-networks are not limited to solving the problem in only one direction.

Finally, conventional spreadsheets that do contain circular references are solved much more efficiently (and reliably) compared to iteration, simply because every iteration step is interpreted while constraint-solving is compiled. Thus, the increase in problem-solving power (compared to iteration) has no cost in performance.

1. As a matter of fact, in a smart implementation one would only build the network only once and incrementally extend and modify it.

V Application and Comparison

After presenting our implementation in the previous chapter we will focus on its problem-solving power in this chapter but not consider any user-interface issues. We give several examples from engineering, physics and business for which the idea of a constraint-based spreadsheet is well, or even uniquely, suited and compare the solution of these examples with what conventional spreadsheets offer to solve the same problem. All examples given are implemented and can be tried out.

V.1 Simultaneous linear equations

The ability to solve sets of simultaneous linear equations without worrying about their exact formulation opens up new application domains for spreadsheets. As we illustrated with **Figure 5** on page 17, in a conventional spreadsheet the user has to (1) recognize the circularity of the problem, (2) turn iteration on explicitly and (3) make sure that all his formulas are of the correct form so that iteration converges. **Figure 13** shows the

Figure 13 The dividend problem solved with the constraint-based spreadsheet

GrossProfit	100
Dividend	
NetProfit	

GrossProfit - Dividend = NetProfit
Dividend = 0.1 * NetProfit

GrossProfit	100
Dividend	9.09
NetProfit	90.91

GrossProfit - Dividend = NetProfit
Dividend = 0.1 * NetProfit

corresponding constraint-based spreadsheet. It is straightforward and has none of the above drawbacks.

The next example is drawn from electrical engineering. Let us assume are given the electrical circuit drawn in **Figure 14.** We set up a spreadsheet that allows us to test this circuit for certain input values. For example, given the resistance for each resistor and given the voltage of the battery we would like to calculate the current and voltage in each component of the circuit. The constraint-based solution is shown in **Figure 15.** We have already used the same example in **Figure 6** on page 27 when introducing vector-

Figure 14 Example of an electrical circuit

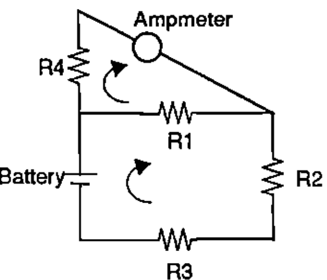
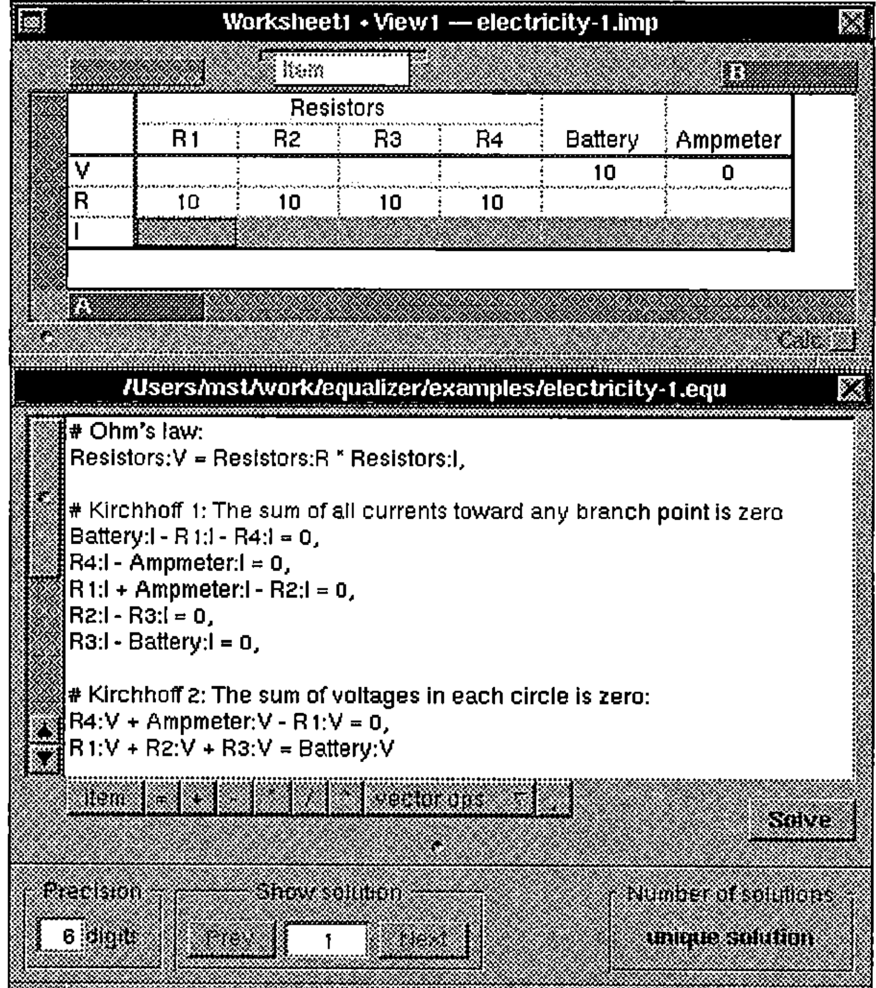


Figure 15 A spreadsheet modelling an electrical circuit



constraints. As we have seen, Ohm's law can be entered as a single vector constraint between the voltage, the current and the resistivity of all **Resistors.** By assuming a direction for the current (see the two arrows in **Figure 14**) Kirchhoff's current law can be formulated as a constraint for each node[1] which states that the sum of the currents entering and leaving any node is algebraically zero. This leads to the five constraints shown.

Similarly, Kirchhoff's voltage law states that the sum of the voltages around any closed circuit or loop in the network is algebraically zero, which gives raise to two additional constraints. As we can see, the basic electrical laws can be entered directly as constraints without transforming them to solve the problem for one particular variable.

V.2 Vector constraints

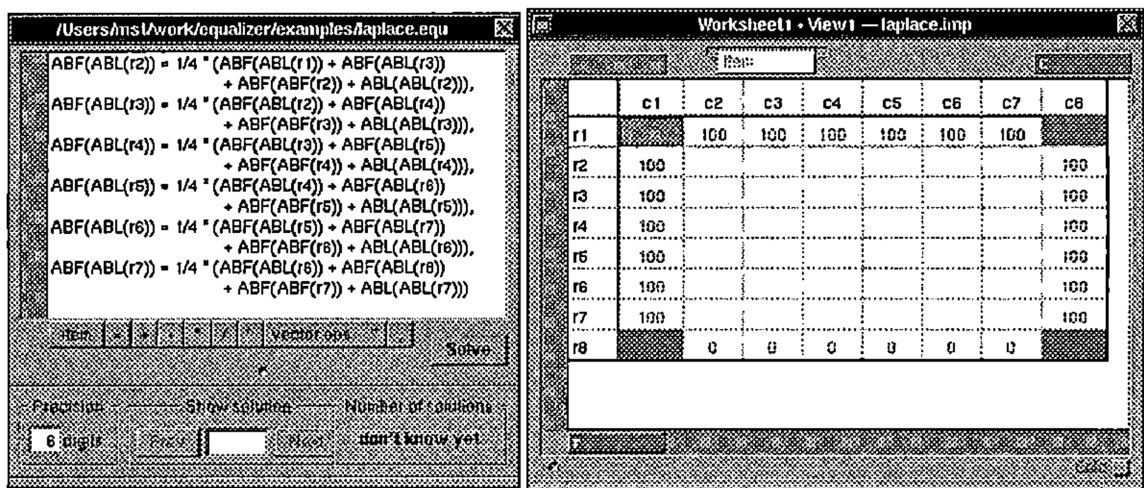
The following problem illustrates the use of vectors and operations on vectors for setting up a large system of linear equations. Consider the flow of heat in a rectangular metallic plate when a heat source is applied to the edge(s) of this plate [6]. The heat from a hot edge will gradually diffuse into the plate and heat up colder regions of the plate until a stationary state is reached. We would like to find the heat at each point of the plate at this stationary state. This classical engineering problem is solved with Laplace's equation. Briefly, in the case of our example Laplace's equation is satisfied when the heat of the plate at each point is the average of the heat at its four closest neighbor points. We use the spreadsheet grid itself to represent the plate and interpret the value of each cell as the heat at that point on the plate. **Figure 16** shows the constraint-based solution to this problem.

V.3 Goal seeking

Most conventional spreadsheets provide a mechanism called *goal-seeking*. Goal-seeking allows the user to give a target or goal value for a cell calculated by a formula and then ask for the value of a cell that is needed for calculating the goal-cell. The same functionality is

1. A *node* in the network is any point to which two or more wires are connected.

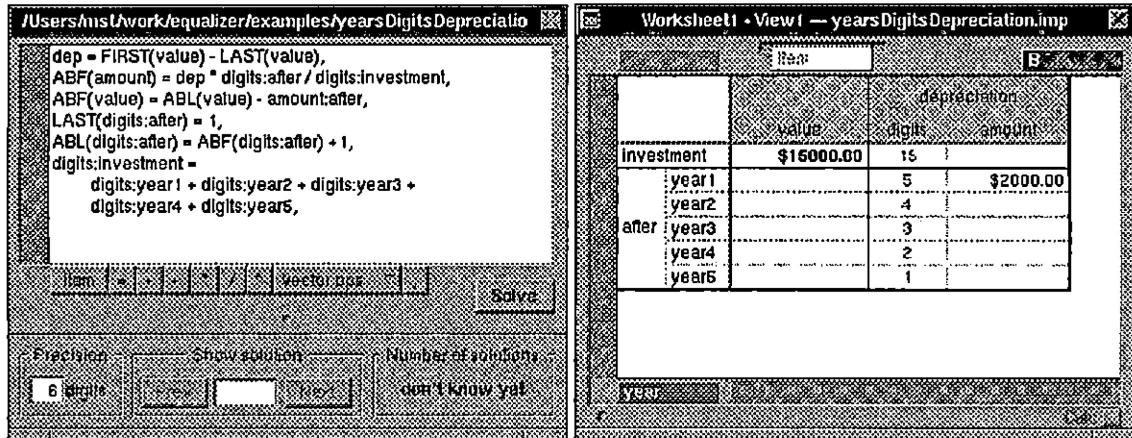
Figure 16 Steady-state two-dimensional heat flow



readily built into a constraint-based spreadsheet. In fact, whichever cell is empty will be calculated from the others. Consequently, the same constraints are good for solving the inverse problem as well. As a first example, in **Figure 15** we can calculate the resistivity of each electrical component and the voltage of the battery, given values the resistor's current and voltage.

Many business problems rely on backward calculation. **Figure 17** shows the spreadsheet that we used in **Figure 7** on page 29 to introduce operations on vectors. The usual application of this spreadsheet is to answer the question: "Given the initial cost the of an investment, its expected lifetime and its salvage value, how much is its depreciation every year". However, without changing any of the constraints nor the spreadsheet itself we can also compute the answer to the problem: "For how much do I have to be able to resell a

Figure 17 Depreciation of an investment according to the year's digit method

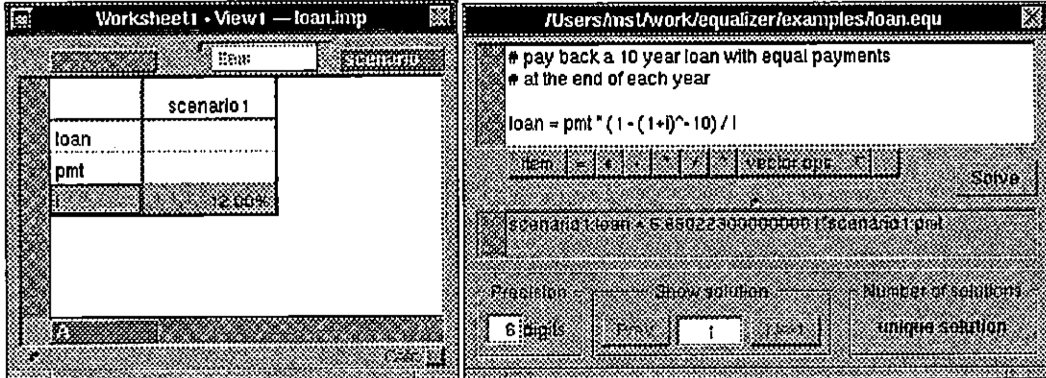


$15,000 investment after five years if I can amortize no more than $2000 the first year". (This is the set-up in **Figure 17**). Or alternatively: "If in three years I can have a (tax-deductible) depreciation of up to $1,000 and at the end of that year the investment is still worth $5,000, how much money can I invest and how much do I have to get back at the end of the five year period for that investment". Or, with the additional constraint:

0.33 * value:investment = amout:year1

and given the cost of the investment we ask: "If the depreciation during the first year is 33%, for how much do I have to be able to resell the investment after 5 years".

Figure 18 Payments at the end of each year to pay back a ten year loan at a given interest rate



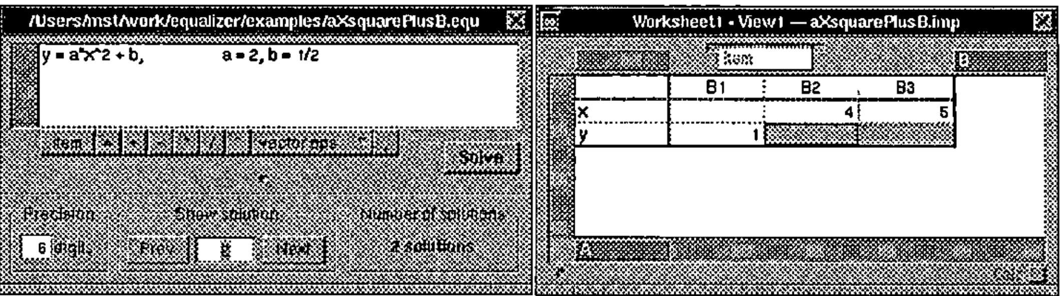
V.4 Under-constrained problems and answer constraints

The following example shows that an answer constraint itself can be the solution to the problem. Consider the spreadsheet in **Figure 17** which calculates payments at the end of each year to pay back a loan over ten years at a given interest rate. If we only provide the interest rate and ask for the loan and the yearly payment we get back as an answer the constraint:

loan:scenrio1 = 5.650223 * pmt:scenario1

The interpretation of this answer is quite interesting. It says that for a ten-year loan at a 12% interest-rate 5.65 payments out of ten (56.5%) are for amortizing the loan while the rest is for the interest.

Figure 19 Multiple solutions



V.5 Multiple solutions

The spreadsheet in **Figure 17** computes several values of the quadratic equations $y = ax^2 + b$. Given a value for y, there are two results for x. The user can browse through all solutions with the "previous" and "next" buttons or enter the number of a solution directly into the field.

VI Conclusion and Future Work

VI.1 Contributions

We have explored, in this thesis, the idea of a spreadsheet based on the concept of numerical constraints between cells instead of formulas. The contributions of this thesis are as follows:

1. We put forward a formalization of the meaning of the typical conventional spreadsheet using denotational semantics. To our knowledge, no work has been done in this direction.
2. Then, we formally define the meaning of our new constraint-based spreadsheet.
3. We introduce an APL-like language for constraints on vectors of cells.
4. We have accomplished a proof-of-concept implementation and
5. we have demonstrated its usefulness with various examples.

We have shown that constraints considerably extend the problem-solving power of conventional spreadsheets without giving up any features. On the contrary, many features that have subsequently been added to conventional spreadsheets, such as goal-seeking and

optimization, are seamlessly integrated into a spreadsheet based on constraints. Moreover, we have shown that the new spreadsheet is compatible with the conventional spreadsheet and we have outlined an implementation where the cost of the increased problem-solving power has to be paid only when needed. As a consequence, existing spreadsheet applications can be reused and made run almost equally efficiently.

VI.2 Future work

Often problems are conveniently expressed as a hierarchy of constraints [17]. A constraint hierarchy consists of a set of required constraints and a set of preferential constraints that do not have to be satisfied but preferably should be. By giving the preferential constraints levels of preference or strengths one can "filter" and sort solutions into "more" and "less preferred".

The idea of connecting a spreadsheet as a user-interface to a mathematics package is quite appealing. For our purpose, we were using only a fraction of the functionality of MATHEMATICA. One could easily make more features of MATHEMATICA accessible through the spreadsheet interface, such as linear programming and differential equations for example. One could further extend the idea and use MATHEMATICA's specialized programming language to program algorithms to solve specific types of problems. Many of these problems currently solved by experts would be made accessible to every-day computer users through the same popular interface they are accustomed to.

References

[1] W. Du, W. W. Wadge, "A 3-D Spreadsheet Based on Intensional Logic", IEEE Software, May 1990

[2] M. Spenke, C. Beilken, "A Spreadsheet Interface For Logic Programming", CHI '89 Proceedings, 1989

[3] M. Spenke, C. Beilken, "PERPLEX: A Spreadsheet Interface for Logic Programming by Example", Research Report, FB-GMD-88-29, Gesellschaft für Mathematik und Datenverarbeitung, November 1988

[4] M. H. van Emden, M. Ohki, A. Takeuchi, "Spreadsheets with Incremental Queries as a User Interface for Logic Programming", ICOT Tech. Rep. TR-144, Oct 1985

[5] F. Kriwaszek, "LogiCalc - A PROLOG Spreadsheet", in B. Kowalski, F. Kriwaszek, "Logic Programming", pp. 105-117, also in: D. Michie and J. Hayes, Machine Intelligence ll, 1987

[6] J. Jaffar, S. Michaylov, P. J. Stuckey, R. H. C. Yap, "The CLP(R) Language and System", Proceedings of the 4th ICLP, 1987

[7] A. Colmerauer, "An Introduction to Prolog III", Communications of the ACM, Vol. 33, No. 7, July 1990, pp. 69-90

[8] R. D. Tennent, "Semantics of Programming Languages", Prentice Hall, 1991, ISBN 0-13-805599-8

[9] S. Wolfram, "MATHEMATICA, A System for Doing Mathematics by Computer", 2nd edition, Addison Wesley, 1991, ISBN 0-201-51507-5

[10] A. H. Borning, "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory", ACM Transactions on Programming Languages and Systems, 3(4), pp. 353-387, October 1981.

[11] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, Vol. 23, No. 1, March 1991

[12] G. L. Steele, "The Definition and Implementation of a Computer Programming Language Based on Constraints", August 1980, Ph.D. thesis, MIT AI-TR.595

[13] W. H. Press, B. P. Flannary, S. A. Teukolsky, W. T. Vetterling, "Numerical Recipes in C, The Art of Scientific Computing", Cambridge University Press, 1988, ISBN 0-521-35465-X

[14] J. H. M. Lee, "Numerical Computation as Deduction in Constraint Logic Programming", Ph.D. dissertation, University of Victoria, 1992

[15] J. Gosling, "Algebraic Constraints", Ph.D. Thesis, Carnegie-Mellon University, May 1983

[16] J. Morris, "MATHEMATICS for Mechanical and Production Engineering", Van Nostrand Reinold, 1981, ISBN 0-442-30459-5

[17] A. Borning, B. Freeman-Benson, M. Wilson, "Constraint Hierarchies", Lisp and Symbolic Computation, January 16, 1992.

A Syntax

```
Program       ::=   Equation
                  | Program ',' Equation
Equation      ::=   Expression '=' EquationTail
EquationTail  ::=   Expression
                  | Equation
Expression    ::=   Factor '+' Factor
                  | Factor '-' Factor
                  | '-' Expression
                  | '(' Expression ')'
                  | Expression '^' Exponent
Factor        ::= | Vector '*' Vector
                  | Vector '/' Vector
                  | Vector
                  | real
Vector        ::=   VHead
                  | 'FIRST' '(' VHead ')'
                  | 'LAST' '(' VHead ')'
VHead               name VTail
```

```
                          | 'LEFT'  '(' VTail ')'
                          | 'RIGHT'  '(' VTail ')'
VTail                       ':' name
                          | <emtpy>
Exponent         ::=        Radix
                            '-' Radix
Radix            ::=        integer '/' integer
                            integer
```

VITA

Surname: STADELMANN Given Names: MARC

Place of Birth: LUZERN Date of Birth: JULY 6, 1964

Educational Institutions Attended:

University of Victoria	1991-1992
University of Geneva	1985-1989

Degrees Awarded:

Licence en informatique de gestion, University of Geneva	1989

Honors and Awards:

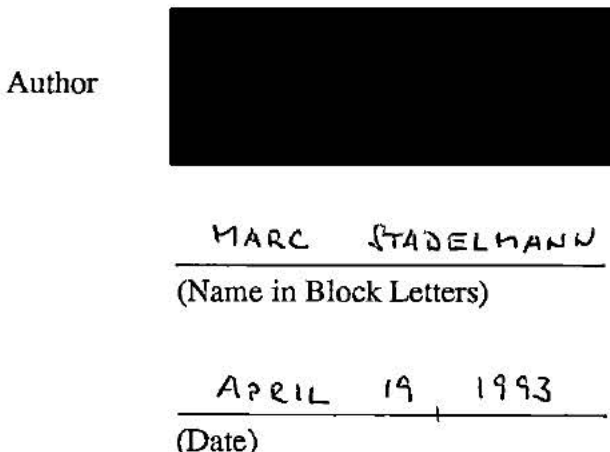
Cap Gemini Prize for second best graduation	1989

PARTIAL COPYRIGHT LICENCE

I hereby grant the right to lend my thesis (or dissertation) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one or its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the university designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

The Design and Implementation of
a Spreadsheet Based on Constraints

Author



MARC STADELMANN

(Name in Block Letters)

APRIL 19, 1993

(Date)