The Knowledge Engineering Review, Vol. 22:3, 221–236. © 2007, Cambridge University Press
doi:10.1017/S0269888907001142 Printed in the United Kingdom

NEXCEL, a deductive spreadsheet

ILIANO CERVESATO

Deductive Solutions, 4903 Regina Drive, Annandale, VA 22003, USA;
e-mail: iliano@deductivesolutions.com

Abstract

Usability and usefulness have made the spreadsheet one of the most successful computing applications of all times: millions rely on it every day for anything from typing grocery lists to developing multimillion-dollar budgets. One thing spreadsheets are not very good at is manipulating the symbolic data and helping users make decisions based on them. By tapping into recent research in Logic Programming, Databases and Cognitive Psychology, we propose a deductive extension to the spreadsheet paradigm that precisely addresses this issue. The accompanying tool, which we call NEXCEL, is intended as an automated assistant for the daily reasoning and decision-making needs of computer users, in the same way as a spreadsheet application such as Microsoft Excel assists them every day with simple and complex calculations. Users without formal training in Logic or even Computer Science can interactively define logical rules in the same simple way as they define formulas in Excel. NEXCEL immediately evaluates these rules, thereby returning lists of values that satisfy them, again just like with numerical formulas. The deductive component is seamlessly integrated into the traditional spreadsheet so that a user not only still has access to the usual functionalities but is also able to use them as part of the logical inference and, dually, to embed deductive steps in a numerical calculation.

1 Introduction

Envisioned in the early '60s, born in the late '70s and popularized during the '80s with the Personal Computer (Power 10/04/2003), the spreadsheet is one of the most ubiquitously used computing applications (it has been surpassed by e-mail programs and Web browsers only in the recent years). Over 50 million people—secretaries, accountants, teachers, engineers, officers, managers and many others—rely on it every day for anything from grocery lists to multimillion-dollar transactions (Boehm *et al.*, 2000). The vast majority of these users have no or little formal Computer Science training, and yet the intuitive interface and clever design of the spreadsheet let them put together complex computations with a few mouse clicks, instantly turning them into unaware programmers. This aspect cannot be overemphasized: in almost no time can a novice user perform simple tasks, and just a few hours of exposure are sufficient to acquire the skills to perform non-trivial computations over large amounts of data[1]. De facto, the spreadsheet is the tool of choice for non-scientific numerical computations and decision-making based on numerical analysis.

[1] The following anecdote illustrates this point well: A friend of the author, physician by trade, discovered the Personal Computer and Excel a few years ago. Within a month, he had entered several years of financial data for all the stocks he owned, together with various formulas to predict their trend. He never found the silver bullet that would allow him to beat the market, but he eventually turned his self-taught Excel skills to Medicine and built a sophisticated application to track the cardiac history of his patients. It is believed that several lives were saved in that way.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

However, a large percentage, if not the majority, of the decisions we make every day depends on the information that is not numerical, or not only numerical. Consider the following simple example from the academic world:

> 'Is Alice, a student who has taken "Operating Systems" and "Theory of Computation," qualified to attend "Advanced Networking"? If not, what other courses does she need to take first? Would that allow her to take "Advanced Networking" next semester? If not, what is the earliest date she can take it?'

Answering each of these questions requires logical reasoning. The last also has a numerical component. Students and courses are symbolic data, and they participate in the relations that describe what courses a student has taken, what the prerequisites of a course are and when courses are offered. Inference rules (e.g. *a student can take a course if he has passed all prerequisites*) describe the reasoning processes that need to be performed to produce the answers. Similar setups and reasoning tasks routinely take place in many professional and personal settings: diagnosis of diseases, legal argumentation, combat readiness of troops, even fashionable color coordination require reasoning patterns akin to the above example.

For all their merits, spreadsheets are pretty much useless when it comes to these forms of symbolic inference: they do offer a few logical operators, but these are designed for simple conditional tests, not deductive reasoning; furthermore, although relations fit snuggly in the familiar row and column layout, the handful of commands that endorse this view have little to do with logical inference. Worse, the computer applications that do support those forms of reasoning, logic programming languages, some databases and many domain-specific tools require a level of sophistication that even many trained programmers do not have. In fact, these applications are seldom installed on an end-user's computing environment. This leaves a wishful problem-solver to either undergo lengthy training or give up on automation. This last option often leads to arbitrary, suboptimal or incorrect decisions.

In this paper, we describe a deductive extension to the spreadsheet that supports sophisticated logical reasoning with the same degree of usability as a traditional spreadsheet application such as Microsoft Excel, hence potentially filling this significant void in the end-user's computing landscape. This *Deductive Spreadsheet* allows users to define logical statements and inference rules for symbolic reasoning in the same way that Excel allows them to define mathematical formulas for numerical calculations. These rules interpret areas of the spreadsheet as logical relations (e.g. a table associating students, each course they have taken and related information) and compute new relations based on them (e.g. what courses a student may take next). Values satisfying them are visualized on the fly and updates anywhere in the spreadsheet are instantly propagated, just as with numerical formulas. As the users may not trust the validity of values obtained in this way, extensive explanation facilities are provided. The Deductive Spreadsheet is a conservative extension of the Traditional Spreadsheet in the sense that every traditional functionality remains available. Furthermore, the deductive extension integrates seamlessly into the existing paradigm in the sense that traditional formulas can participate in logical inferences, and deduced values can be used in arithmetic calculations.

Under the hood, the added deductive capabilities are provided through machinery borrowed from the field of Logic Programming (Lloyd, 1987), but adapted to the peculiarities of the spreadsheet environment and the usability requirement of our target audience: those 50 million managers, officers, engineers, teachers, accountants, secretaries, *etc*., who do not have a formal training in either Computer Science or Logic, and yet would welcome unintimidating automated assistance with their daily reasoning tasks. We use a small logic programming language inspired by Datalog (Lloyd, 1987) to define derived relations: the target region of the spreadsheet contains a set of logical clauses in the same way that calculated cells contain a numerical formula in the Traditional Spreadsheet. Therefore, logical reasoning reduces to computing tables of data by evaluating Datalog-like definitions, a process that parallels the calculation of numerical formulas.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

Each row in the calculated relation is a tuple of values satisfying the definition for this relation, so that the evaluated table lists all such solutions, without repetitions.

We based our language extension on Datalog for two reasons. First, it admits the evaluation strategies that are always bound to terminate, a property that does not hold of other logic programming languages such as Prolog (Lloyd, 1987). This is rather important considering our target audience: although expert Prolog programmers know how to debug infinite loops, end-users experience non-termination only when an application hangs, an event often followed by a reboot. Second, one such (terminating) strategy, bottom-up evaluation, closely fits the intended mode of operation of the Deductive Spreadsheet as it explicitly computes and maintains the values of all defined Datalog clauses. Other applications of Datalog, in particular in the context of Deductive Databases (Ceri *et al.*, 1990), are query-oriented and often take advantage of more focused strategies. Within the Deductive Spreadsheet, we rely on a second strategy, a terminating variant of top-down evaluation, as a basis for an explanation mechanism that permits users to inquire about why a particular entry appears in a calculated relation. We extended the generally accepted definition of Datalog to provide an interface to the language of numerical formulas of the Traditional Spreadsheet, and also to extend the expressiveness of that language to solve a larger number of practical problems. The resulting dialect of Datalog incorporates negation, constraints, calculated values and flat lists. This extension has been engineered in such a way as to maintain all the desirable properties of the original language.

The linguistic extensions just described are made available to the users by means of a parallel extension of the user interface of the Traditional Spreadsheet. Our main goals in that respect were to provide these extended functionalities in a minimally intrusive way so that users who do not want to take advantage of them are not penalized, neither cognitively nor from a performance point of view, and that users who do want to use them can acquire the skills gradually by experimenting with familiar concepts. At first sight, the result is nearly indistinguishable from the layout of a traditional spreadsheet application. However, once a user selects a range of cells, he/she can enter Datalog clauses using an extension of the current methods for creating numerical formulas: a slightly beautified Datalog syntax allows typing them in directly, they can be constructed by clicking and dragging spreadsheet entities with the mouse, and they can be defined using dedicated wizards. In all cases, sophisticated form of visual feedback and precise error-reporting assist the user in this task. Changes to formulas and the data they reference are propagated instantly. Most modifications to primitive and derived relations are handled transparently, including the cut and paste of rows, but certain changes to the geometry of these tables, for example, deleting a column, require the intervention of the user. Explanation facilities are provided for debugging and auditing purposes. In particular, a dialog akin to a directory browser allows a user to inspect the reasoning process that caused a tuple to appear in the result, step by step. The possibility of performing logical reasoning enables support for a number of convenient new productivity tools, such as connection graphs or workflow graphs.

As we were designing the user interface of the Deductive Spreadsheet, we heavily relied on recently proposed methodologies from the field of Cognitive Psychology, notably the Cognitive Dimensions of Notation (Green & Petre, 1996) and the psycho-economic Attention Investment Model (Blackwell, 2002). These approaches have been used with great success in the development of other extensions of the Traditional Spreadsheet (Peyton Jones *et al.*, 2003). We performed a small-scale and very preliminary user evaluation of the interface and the extended functionalities. We obtained positive feedback and a few suggestions for improvement from the intermediate and advanced users, who we were mainly targeting. To our surprise, we also got a good feedback from the beginners we tested, although they clearly struggled with some of the more advanced concepts.

We are in the process of implementing a Deductive Spreadsheet prototype as an add-on module to Microsoft Excel 2003. The resulting system, which we call NEXCEL, is primarily meant as an evaluation testbed for the concepts presented here, in preparation for a commercial-strength reimplementation, either as an extension of an existing product or as a standalone application.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

This work constitutes an extension to the spreadsheet paradigm itself, and therefore can be realized in any commercial application.

The remainder of this paper outlines the development of our proposal for a Deductive Spreadsheet. Section 2 briefly examines the key concepts underlying the traditional spreadsheet and recalls the various attempts proposed throughout its history to support logical reasoning. Section 3 surveys the main extensions to the core functionalities of the Traditional Spreadsheet, while Section 4 examines the corresponding extensions to the user interface. In Section 5, we report on the feedback we obtained from users in a preliminary evaluation. Section 6 gives an overview of the ongoing implementation effort. Although our design employs sophisticated techniques from Logic Programming, Database Theory and Cognitive Psychology, we will strive to maintain our presentation at an intuitive level. Readers interested in the technical aspects of this work are referred to Cervesato (2005).

2 Spreadsheets and beyond

In this section, we identify some of the key concepts underlying the Traditional Spreadsheet, and summarize extensions with deductive capabilities proposed in the literature. In Section 2.1, we outline a simple model of the core functionalities of a spreadsheet, relying mainly on examples—a detailed technical treatment can be found in Cervesato (2005). Then, in Section 2.2, we briefly review the historical proposals for extending the Traditional Spreadsheet with logical inference.

2.1 The traditional spreadsheet model

To most people, a spreadsheet is a grid of cells in which one enters numbers, strings and arithmetic formulas, such that the latter are automatically calculated. We will now explore the various objects that constitute a spreadsheet: the basic scalar infrastructure, array formulas and the little relational support available in commercial products. We will emphasize some of the lesser known entities that we will use as inspiration, and sometimes building blocks, for our deductive extension in Section 3. We also discuss the principal functionalities supported by a spreadsheet application: evaluation, update and explanation. We delay the discussion of user-interface issues until Section 4.

When talking about spreadsheets, we shall draw a distinction between what the user types using his/her keyboard (or enters in other ways) and what he/she sees on his/her screen. We refer to the former manifestation as a ‘spreadsheet’, and use the name ‘evaluated spreadsheet’ for the latter. A spreadsheet in the first sense is therefore a syntactic entity, and as such it is saved to a file, for example. An evaluated spreadsheet is the result of a semantic operation, in particular evaluation or update propagation. As we model the spreadsheet paradigm and the proposed extensions, we will be interested in both aspects: the (syntactic) way things are written, and the operations that interpret this syntax and visualize the values the user is ultimately interested in.

2.1.1 Scalar spreadsheets

We begin with a model that reflects just the quintessential aspects of the spreadsheet paradigm, ignoring advanced features such as array formulas for the moment. We call this minimal model a ‘scalar spreadsheet’: although it falls short of what commercial applications offer, it captures what a spreadsheet is to most users.

From a syntactic point of view, a (scalar) spreadsheet is a collection of cells that can contain either values or formulas. Examples of *values* are the numbers ‘42’ and ‘12.99’, or the string ‘Total’ (dates and currencies are just numbers visualized in a special way by means of formatting directives). Most cells in a typical spreadsheet contain the default *blank* value.

A cell can also contain a *formula* such as ‘$=(A3-32)*5/9$’: after evaluation, this cell will display the result of taking the contents of the cell *A3*, subtracting 32 from it, multiplying the result by 5 and dividing it by 9. A (scalar) formula is constructed out of values (e.g., ‘32’, ‘5’ and ‘9’), *operators* (here, ‘−’, ‘*’ and ‘/’)and *cell references* such as *A3* in this example. The symbol ‘=’ is

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

commonly used to distinguish formulas from values (strings in particular); we will omit it hereafter. The cell reference '*A3*' identifies the cell on column *A* and row 3 of the current worksheet (the exact syntax for references is not important for our purposes, and indeed other notations exist). Commercial applications support spreadsheets consisting of several worksheets, each with typically 65,536 rows and 256 columns—these numbers are again unimportant from a modeling standpoint.

In this early model a spreadsheet is therefore a collection of cells, each of which contains either a value or a formula. Valid spreadsheets are subject to one condition: cells are not allowed to contain *circular dependencies*, that is, a cell *c* cannot contain a formula that references *c*, directly or indirectly. For example, the above formula cannot be inserted in cell *A*3, in any cell possibly referenced in cell *A*3 (if *A*3 contains a formula), in any cell referenced in a cell referenced in cell *A*3, *etc*[2]. From a programming language point of view, a spreadsheet is a simple functional language without recursion.

Of course, a user is not interested in formulas per se, but in the values computed by them. Spreadsheet applications provide three core functionalities for going back and forth between a (syntactic) spreadsheet and an evaluated spreadsheet: evaluation, updates and explanation.

When loading a spreadsheet file (or pressing a special key combination), a spreadsheet application computes the formula contained in each cell and displays the corresponding value. This process is called *evaluation* and the result is the evaluated spreadsheet displayed on the user's screen. Intuitively, evaluation starts with the cells, call them C_0, hich contain a value in the original spreadsheet and displays them unchanged. It then examines those cells, say C_1, containing formulas that only reference cells among C_0 and computes their value. It continues with the cells C_2 that reference at most cells in C_0 and C_1, and so on until all the cells have been given a value. Because of the absence of circular dependencies, this process always terminates with the evaluation of the entire spreadsheet. It can be efficiently performed in a time proportional to the number of nonblank cells. The partially evaluated spreadsheet obtained at each step is called an *environment*: evaluation starts from a totally undefined environment and progressively populates its cells with values. The evaluated spreadsheet is then the final environment, which does not contain any undefined cell. The interested reader is referred to Cervesato (2005) for an algorithmic description and a detailed analysis of evaluation.

Full evaluation is a very rare event that takes place mostly when loading a spreadsheet from file. Much more common is *updating* a spreadsheet, something that happens every time the user modifies the contents of a cell. *Recalculation* is the process of propagating changes to all parts of the spreadsheet that depend on the modified locations by updating the displayed values of the affected cells. Although recalculations can be implemented as evaluation, it is almost invariably much more efficient to identify the cells affected by the changes (either because its contents have been modified or because they depend on a modified cell) and re-evaluate only their contents. This can be efficiently achieved by invoking a form of partial evaluation that starts from an environment that simply marks all cells affected by the update as undefined, and preserves the former value of every other cell.

Explanation is the ability to answer questions such as '*why does this location report this value?*'. It has been recognized as one of the essential functionalities of a spreadsheet application, as it has been shown that the vast majority of spreadsheets are riddled with errors of all types (Panko, 1998). Formulas are a common source of errors, especially when they reference other cells. Commercial spreadsheets support explanation by offering tools that visualize dependencies among cells in the evaluated spreadsheet. By using these tools, a user can highlight the cells that are referenced

[2] To be fully precise, most commercial spreadsheet applications allow advanced users to enable a restricted form of circular dependency which permits calculating numerical approximations of the solutions of some types of equations using bounded iterations. While our model and the proposed extension can easily accommodate this rarely used feature, doing so would be an unnecessary distraction from the main focus of this paper.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

in the formula contained in a cell, and chase such dependencies forward or backward often until an error is discovered.

2.1.2 Arrays

Arrays, or *cell ranges*, are a familiar concept to spreadsheet users. For example, the simplest way to add all the values in column A from row 2 to 31 is to enter the formula SUM($A2 : A31$) into a cell: here, $A2 : A31$ is an *array reference* and SUM is an operator that accepts an array as an argument. Not as well known is the fact that most commercial spreadsheets make available a mechanism by which a formula can *evaluate to* an array, that is, produce a value not just for a single cell but also for multiple cells at once. These are called *array formulas*. In Microsoft Excel, this can be achieved, for example, by selecting cells $C1 : C3$, entering the formula $A1 : A3 + B1 : B3$ and sealing it with the special 'CTRL-SHIFT-ENTER' key combination. This will have the effect of adding the contents of arrays $A1 : A3$ and $B1 : B3$ component-wise and displaying the results in cells $C1$ through $C3$. Mismatches between the argument geometries (e.g. adding a 3-element array to a 5×7-element array) are automatically adjusted through default expansion and truncation rules.

Array formulas differ from the scalar expressions described earlier by the fact that they reside not in a single cell but in a range of cells such as $C1 : C3$ in the above example. In order to capture this aspect, we need to upgrade our notion of (syntactic) spreadsheet: not only individual cells but also groups of cells can hold a formula. Clearly, any given cell can be associated to at most one array formula in this way (or it can hold a value or a scalar formula). This possibility, of writing an expression in a range of cells, is one of the pillars of the extension proposed in this paper: note again that it is a standard (although seldom used) feature of any commercial spreadsheet. In Section 3, we will assign to such ranges not traditional array formulas but inference rules that permit logically deriving sets of values from the data present in other parts of the spreadsheet.

From a semantic standpoint, the array operators available in commercial spreadsheet applications are such that array expressions can always be simulated with scalar formulas: arrays introduce convenience, but no extra expressive power. In particular, any spreadsheet (with array formulas) has a canonical scalar counterpart (the above example is simulated by entering the scalar formulas $A1 + B1$, $A2 + B2$, $A3 + B3$ in cells $C1$, $C2$ and $C3$, respectively). Furthermore, the semantics of a generic spreadsheet is defined on the basis of the semantics of this canonical scalar representative. Not only are evaluation, update and explanation defined in this way but also is the very notion of dependency. Indeed, the apparently circular insertion of the array formula $A1 : A5 + A2 : A6$ in the range $A3 : A7$ is legal because its scalar expansion does not have circular dependencies.

Although logical rules reside in the same cell ranges as array formulas, the deductive extension proposed in this paper cannot be simulated by current scalar formulas, in general. Therefore, it represents a genuine (and significant) increase in expressive power with respect to the Traditional Spreadsheet.

2.1.3 Relational support

The tabular layout of a Traditional Spreadsheet naturally lends itself to interpreting groups of columns as *relations*. For example, columns A through D in a worksheet may be used to record student names, classes, the grades of each student for the corresponding class, and the dates they were awarded. A row is then seen as a *record* of related values, for example, '(Alice,OperatingSystems, A −,12/14/2005)'. The values in each participating column are understood as having the same meaning.

The deductive extension in Section 3 will heavily rely on the notion of relation, or more precisely on its logical counterpart, the concept of predicate. Indeed, inference rules will generally compute to relations, that is, upon evaluation, they will fill the cell range in which they are defined with a set of records. In order to do so, they will typically interpret other areas of the spreadsheet as relations and combine portions of their records into their own result. Section 3 will be mainly concerned with defining the language of these logical formulas and how to evaluate them, while Section 4 will describe how to conveniently make them available to users.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

It should be noted that commercial products do provide a limited toolkit for supporting the relational interpretation of a region of the spreadsheet. Microsoft Excel, for example, recognizes this intent when the user enters textual captions in the top row of a new worksheet and then inputs the first record. Excel then interprets this group of columns as a 'data list' and makes available a number of commands to manipulate it: it defines a 'form' to input new records or modifying existing ones; it offers 'filters' to hide from view all the records that do not satisfy a given condition; it permits sorting a data list in place; it provides a number of 'database functions' to conditionally perform operations such as SUM on a data list; finally, it allows a set of columns to be imported from an external application, either a database or spreadsheet—this is the only non-manual way of creating a relation.

This relational support is limited and limiting in a number of ways. First, it lacks completeness as few rudimentary operations are available to combine two or more relations (the most interesting are VLOOKUP and HLOOKUP, which allow simulating at best some cases of outer joins, with much effort, but not the more useful inner join). Second, this support is mostly provided in the form of commands (like 'Print', say) rather than operators (like '+') that can be woven into formulas. This endows the relations with a second-class nature that sits on top of but does not blend into the spreadsheet paradigm: relational command are not bound to the relations they compute, they must be initiated by the user rather than performed automatically through calculation, and updates are not automatically propagated.

By contrast, our effort makes available a linguistic extension that allows working with relations (or more accurately predicates) as first-class objects in the same way as current formulas operate on cells: the logical rules defining a predicate are bound to a cell range exactly like array formulas. In particular, they are saved to file just like any formula, they are evaluated upon loading a (deductive) spreadsheet from a file, and they immediately propagate any update affecting the data they depend on.

2.2 Related work: mixing spreadsheets and logic

We will now review the main historical attempts at infusing spreadsheets with deductive capabilities.

2.2.1 1982–2004

The idea of combining logic programming and spreadsheets started circulating in the early '80s, at a time when both technologies were relatively new, and the personal computer had just begun trickling off the production lines. The first and maybe most comprehensive early proposal was put forth by Frank Kriwaczek in his 1982 Master's thesis (Kriwaczek, 1982), a revised version of which was later published for a wider audience as (Kriwaczek, 1988). His LogiCalc system was meant to 'reconstruct logically, in micro-Prolog, a typical spreadsheet program'. It not only captured most of the functionalities of the early spreadsheets but also provided powerful extensions such as relational views, integrity constraints, bidirectional variables, symbolic manipulation, and complex objects (e.g. lists, graphics and spreadsheets themselves). The input and output was based on the limited teletype interface of early computers, as modern graphical user interfaces had not been invented at the time.

A few years later, van Emden proposed using the concept of incremental query, by which a standard Prolog query can be refined interactively, for solving spreadsheet-like problems (van Emden *et al.*, 1986). The proof-of-concept Prolog implementation realized core spreadsheet functionalities as well as exploratory logic programming, but made use of the (still non-graphical) matrix display of the spreadsheet only for output, one solution at a time. This ideas was later refined (Cheng *et al.*, 1988) to report all solutions to a Prolog query in a tabular fashion using a windowed graphical interface.

The last attempt of the decade at integrating logic and spreadsheets was Spenke and Beilken's PERPLEX system (Spenke & Beilken, 1989), which relied on logical formulas to express

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

bidirectional integrity constraints within a fully graphical spreadsheet. The resulting system was very powerful but unlikely to be usable by an entry-level user.

Surprisingly, this thread of research dried out in the late '80s in spite of the gigantic advances in user interface design and the coming of the age of logic programming in the '90s. The one isolated exception was the Knowledgesheet system (Gupta & Akhter, 2000), which explored an alternative way of embedding logical constraints within a spreadsheet.

2.3 Recent efforts

In a sharp reversal of this trend toward oblivion, half a dozen logical extensions to the spreadsheet paradigm were proposed in 2004–05. This flurry of activity culminated in the Workshop on Logical Spreadsheets—WOLS'05 held at Stanford University in the summer of 2005 (*Workshop on Logical Spreadsheets*, 2005). The presented systems fell into two clear classes: proposals that allowed the Prolog-style computation of new values from existing values, and systems that provided support for bidirectional constraints. The present work falls into the first class.

3 Extending core functionalities

We will now introduce the functional extensions underlying the Deductive Spreadsheet. As mentioned earlier, our objective is to make support available for manipulating relations as first-class objects, namely to write formulas that compute to relations and to extend the traditional mechanisms for evaluation, update and explanation to these new entities. As our language for relational formulas, we designed a variant of the logic programming language, Datalog (Lloyd, 1987; Ceri *et al.*, 1990). Datalog has been studied for many years in the context of Deductive Databases (Ceri *et al.*, 1990; Colomb, 1998); it has excellent computational properties and expressiveness; and it comes with a number of traditional algorithms that are compatible with our intended use in a spreadsheet context. We extend its syntax to support embedding the array and scalar formulas of the Traditional Spreadsheet and to make it usable in a wider range of practical circumstances. Supported algorithms are extended accordingly. We will now outline the salient aspects of this program, relying on examples for the most part. Formal definitions and detailed analyses can be found in Cervesato (2005).

3.1 First-class relations and logical infrastructure

Assume that a tabular area of a spreadsheet has been loaded with the information about commercial flight segments, for example, using the 'data list' forms available in the Traditional Spreadsheet, or by some other input method. In order to make this example more concrete, suppose that columns *A*, *B* and *B* contain the origin and destination airports of every such flight, and the distance between them, respectively. Figure 1 partially displays this setting within Microsoft Excel 2000 (please ignore the textbox and menus for the time being), with the addition of captions for the whole table ('*directFlight*') and the individual columns ('*From*', '*To*' and '*Dist*'). For the sake of readability, we will use these names to refer to the actual table and its columns, instead of cell ranges such as $A3 : C200$ for example (commercial spreadsheet provides means to associate symbolic names to cell ranges and other spreadsheet entities).

We now want to compute all airport pairs that are reachable by flying through at most one intermediate city. Intuitively, they consist of all pairs of cities that can be reached in either one hop or two hops. The first option is just the table directFlight, with the distance information omitted: if one can fly directly from A to B, then one can fly in at most two hops from A to B. The second way is to fly to an intermediate city and from there to a destination: if one can fly from A to X directly and then from X to B directly, then one can fly in at most two hops from A to B. This simple reasoning pattern provides a way of building the desired relation, call it

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

twoHopsMax. The logic programming language Datalog (Lloyd, 1987; Ceri *et al.*, 1990) formalizes such patterns and provides a syntax to express them. This simple example assumes the following form in Datalog:

$$\begin{aligned}&\text{twoHopsMax(A,B)} \leftarrow \text{directFlight(A,B,_)}.\\&\text{twoHopsMax(A,B)} \leftarrow \text{directFlight(A,X,_)} \wedge \text{directFlight(X,B,_)}.\end{aligned}$$

Here, the *predicate* directFlight(A,B,_) describes a generic record of the table directFlight, using the *variables* 'A' and 'B' to identify the values in its first and second column, respectively; the notation '_' is a way to ignore the third value in this record. Then, it suffices to read the reverse implication symbol, '$\leftarrow$', as 'if' and the conjunction symbol, $\wedge$, as 'and', and we obtain an exact reading of the above informal definition of twoHopsMax: the formula on the first line states that one way to build a record for twoHopsMax is to take any tuple from directFlight and ignore the third component; the second formula says that one can alternatively combine the first and second value in two arbitrary records, as long as their second and first values coincide. Each of these formulas is called a *clause* in Datalog, and altogether they contribute to defining the predicate twoHopsMax.

Now, assume that we want this derived relation to be written in columns D and E of our spreadsheet, evaluated when loading it from a file and automatically updated whenever any value directFlight in changes. There is no way to achieve this effect within the Traditional Spreadsheet.

By contrast, the Deductive Spreadsheet allows associating a clausal definition such as the above to a region of the spreadsheet. Indeed, we can select the cell range $D3 : E500$ and assign these two clauses to it, in nearly the same way as we would install an array formula into it. Again, we use symbolic names for this table and its columns. Evaluating these clauses will fill this table with unique records satisfying them, starting from row 3. If there are less than 497 such records, the bottom portion of the table will remain blank; if there are more, some entries will be dropped (the user interface provides a notation for ranges such as columns D and E from row 3 to the bottom of the worksheet).

The Deductive Spreadsheet provides syntax to define a wide range of relations. Let us first modify the above example to return only those pairs of cities that are at most 200 miles apart. This is achieved by the following two clauses:

$$\begin{aligned}&\text{twoHopsMax(A,B)} \leftarrow \text{directFlight(A,B,D)} \wedge \text{D} < 200\\&\text{twoHopsMax(A,B)} \leftarrow \text{directFlight(A,X,D)} \wedge \text{directFlight(X,B,D}') \wedge \text{D} + \text{D}' < 200\end{aligned}$$

The added conjuncts, $\text{D} < 200$ and $\text{D} + \text{D}' < 200$, are called *constraints*. Constraints are one of the available interfaces to embedding traditional spreadsheet formulas within a clausal definition. Indeed, 200 here is just a value, while the expression $\text{D} + \text{D}'$ is closely related to scalar formulas. In general, a constraint can compare any expression built out of values, operators, cell references and clausal variables. The other way to refer to a traditional formula within a clause is to use the *coercion operator* $\langle \ldots \rangle$, which converts a generic array formula to a relation with the same geometry (the dual [...] coercion interprets a clausal definition as an array for inclusion in a traditional formula).

The above examples make a fairly basic use of the logical capabilities of Datalog. The following recursive clauses compute the pair of airports that can be reached via arbitrarily many hops:

$$\begin{aligned}&\text{indirect(A,B)} \leftarrow \text{directFlight(A,B,_)}.\\&\text{indirect(A,B)} \leftarrow \text{directFlight(A,X,_)} \wedge \text{indirect(X,B)}.\end{aligned}$$

Notice that *indirect* is defined in terms of itself: there is an itinerary from A to B if there is a direct flight between them, or if there is a direct flight from A to some intermediate destination X and an itinerary from X to B.

Datalog allows other logical operators to appear in clauses, in particular a restricted form of negation called *stratified negation* (Lloyd, 1987; Ceri *et al.*, 1990). One thing that Datalog does

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

not permit is for a clause to return values computed by applying a functional operator to recursive argument (for instance, adding an argument to indirect to keep track of the number of segment): this is unsafe as it may cause evaluation never to terminate. From a practical point of view, computing values in this way is, however, extremely useful: one does want to know the number of hops in his itinerary. The Deductive Spreadsheet accommodates this need by permitting such calculations, but stopping the computation after a user-defined bound on the maximum number of iterations. It is then possible to compute the number of premier (25% bonus) frequent flyer miles an itinerary will yield, which would be forbidden in Datalog:

$$\begin{aligned}&\text{indirect}(\text{A},\text{B},\text{M}) \leftarrow \text{directFlight}(\text{A},\text{X},\text{D}) \wedge \text{M} = 1.25 * \text{D}.\\&\text{indirect}(\text{A},\text{X},\text{M}) \leftarrow \text{directFlight}(\text{A},\text{X},\text{D}) \wedge \text{indirect}(\text{X},\text{B},\text{M}') \wedge \text{M} = 1.25 * \text{D} + \text{M}'.\end{aligned}$$

We similarly stretch the traditional limits of Datalog by allowing flat lists, as in the following example which returns not just pairs of airports that are linked by air but also the actual itinerary between them:

$$\begin{aligned}&\text{indirect}(\text{A},\text{B},\text{It}) \leftarrow \text{directFlight}(\text{A},\text{B},_) \wedge \text{It} = [\text{A},\text{B}]\\&\text{indirect}(\text{A},\text{B},\text{It}) \leftarrow \text{directFlight}(\text{A},\text{X},_) \wedge \text{indirect}(\text{X},\text{B},\text{It}') \wedge \text{It} = [\text{A}|\text{It}'].\end{aligned}$$

Bounded iteration is again used to avoid the possibility of an infinite computation.

Complex deductive patterns can be achieved by appropriately combining the logical features introduced so far. For example, it is possible to find the shortest itinerary between two cities, the number of such itineraries, the list of all airports reachable from a given city, *etc*. Abbreviations are available for many of these advanced patterns. The resulting language allows expressing numerous classes of problems that are not solvable in the Traditional Spreadsheet. Examples include the bill of materials, anti-trust control, meeting planner, workflow and various types of transitive closure (see Cervesato, 2005 for details).

3.2 Evaluation and updates

Datalog has been extensively studied in the context of databases, as its recursive clauses extend the expressiveness of the traditional query languages of Relational Databases (Ceri *et al.*, 1990; Colomb, 1998). Particularly appealing is the fact that, differently from other logic programming languages like Prolog, there are algorithms which guarantee that the evaluation of a set of Datalog clauses will always terminate. This is of prime importance in the context of a spreadsheet as the prospect of non-termination would not be accepted by users.

A number of (terminating) evaluation algorithms have been proposed for Datalog (Ceri *et al.*, 1990). Many of them are optimized for efficiently answering typical database queries, which tend to be rather focused, returning just a few records. This is not the way we anticipate the deductive extension of the spreadsheet will be used. Instead, we expect that the typical user will define rather general relations, similar to the examples given above, characterized by a number of records comparable to the tables they draw their input from. One of the earliest proposals for evaluating Datalog clauses, *bottom up evaluation*, is particularly suited to this mode of operation as it efficiently computes sets of records until all possible solutions have been produced. In order to produce the set of values satisfying indirect, for example, it would first return the pairs of cities that are directly linked (using directFlight in the first clause), then those that require one stop, then two stops and so on. At each step, records previously obtained are discarded; the process ends when no new record is generated in this way. We have extended this approach to encompass the language outlined in the previous section, including the aspects that are not found in Datalog (Cervesato, 2005).

An optimized version of this evaluation method, known as the *semi-nave strategy*, forms the basis of an efficient procedure for update propagation. While updates in the Traditional Spreadsheet statically identify the affected cells, this method attempts to dynamically identify the records

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

that should be added or deleted to a derived relation. This can be done quite efficiently for common types of clausal definitions. We invite the interested reader to consult Cervesato (2005) for details.

3.3 Explanation

Explanation in the Traditional Spreadsheet boils down to following cell references in formulas, which is reasonable since the result of a formula typically depends on all the values that are referenced in it. This is not so in our deductive extension: although the definition of indirect references the relation directFlight, for example, any given record in the former is actually computed on the basis of very few records in the latter. Therefore, just following static references is not particularly useful in this new setting. In fact, the type of questions we are interested in is not so much '*What does this relation depend on?*', but rather '*Why was this record returned?*'. This requires unfolding the computation that produced this particular record, not tracking down static dependencies.

One way to answer such questions is to keep track of how each record was obtained. This approach not only requires considerable bookkeeping, but is ineffective vis-a-vis of another legitimate type of questions: '*Why was this record not returned?*'. The explanation facilities of the Deductive Spreadsheet operate differently: it starts with the suspect record (or in general an arbitrary query) and unfolds the available clauses in search of all supporting facts in the spreadsheet (or their absence). This technique employs a *top-down* strategy as it unravels definitions from the top query all the way down to elementary evidence. Unless conducted carefully, top-down evaluation can be non-terminating even in Datalog. The Deductive Spreadsheet relies on the approaches that guarantee termination (Smith *et al.*, 1986; Warren, 1998).

4 User interface extension

The Traditional Spreadsheet owes much of its success to a user interface that has evolved to provide intuitive access to the underlying functionalities, for the most part (array formulas are an unfortunate exception). In particular, it offers consistent interaction modalities across features and a gentle learning curve that allows users to progress through exposure and minimal experimentation. One of our main goals in the design of the Deductive Spreadsheet was to maintain this very same level of usability as we made available the added expressiveness of logical reasoning. For this reason, we relied on the user interface design methodologies proposed recently within the field of Cognitive Psychology and applied with great success in other extensions of the spreadsheet (Peyton Jones *et al.*, 2003). Using them as guidelines, we developed a minimally intrusive extension of the user interface of the Traditional Spreadsheet to support the deductive functionalities described earlier. It extends all traditional methods for interacting with the spreadsheet to the new deductive features, in particular as far as formula manipulation is concerned. We will now briefly review both the interface design approach and the outcome of adopting it. Details can be found in Cervesato (2005).

4.1 Approach

Following the steps of Peyton Jones *et al.* (2003), we used two Cognitive Science methodologies to design our initial user interface of the Deductive Spreadsheet: the Cognitive Dimensions of Notation (Green & Petre, 1996) and the Attention Investment Model (Blackwell, 2002). The first relies on the observation that the designers of a system, language or computer interface often lack names to talk about some of the cognitive concepts they use, especially those concepts that most directly impact the end user. It then proceeds to providing a concrete vocabulary to bring these often implicit concepts to the foreground and help make informed user-centered decisions. For example, it defines 'premature commitment' as the degree to which a user is forced to make a decision before all the information is available. By contrast, the Attention Investment Model

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

(Blackwell, 2002) '*offers a cost/benefit analysis of abstraction use that allows us to predict the circumstances in which users will choose to engage in [them]'*, hence encouraging the designer to put himself in the shoes of the user and anticipate how they will cope with aspects of the notation.

4.2 Realization

Our primary concern in designing the user interface of the Deductive Spreadsheet has been to remain *conservative* with respect to the choices that have sedimented in the mostly excellent interfaces of modern spreadsheet applications, choices to which users have become accustomed and sometimes dependent. Within these necessary bounds, our major design objective has been to provide the user with a simple and intuitive access to the enhanced expressive power of the deductive infrastructure. For the most part, this is realized by simply extending the current interaction modalities to the new deductive components. For example, all the graphical approaches to constructing a scalar or array formula are available when building a clausal definition. We occasionally extend current support in order to further simplify the user's experience, in particular by offering a new interactive way to assemble clauses or to take advantage of the deductive infrastructure, for instance, by providing novel flow graph visualization mechanisms.

At first glance, the user interface of the Deductive Spreadsheet is indistinguishable from the screen of a typical Traditional Spreadsheet. Indeed, the user still sees a grid of cells which he/she can format in the usual ways. He/she forms a worksheet and several worksheets constitute a workbook. All the traditional navigation methods, selection techniques and commands at large operate as usual. It is only when he/she starts working with the system that the new possibilities reveal themselves as added menu items and extensions to old functionalities. The major changes to the static layout of the application window cater around providing simple and efficient methods for designating areas of the worksheet as relations, and making a convenient interface available for associating a clausal theory with a range of cells. Although the first objective can be realized by chaining commands in the traditional spreadsheet, we streamline this process by providing a simple dialog for achieving it.

The second objective comprises entering clausal definitions and invoking deductive capabilities. We currently offer two equivalent syntaxes for writing clauses: one is a slightly beautified form of Datalog (see Figure 1 for an example) and the other extends the traditional SQL query language of relational databases—see Cervesato (2005). Our main motivation for this dual language approach is that some users may find it easier to express their problem logically, while others may have more of a relational mind-set. The textual syntax of logical formulas is, however, likely to evolve as we gather feedback from the user.

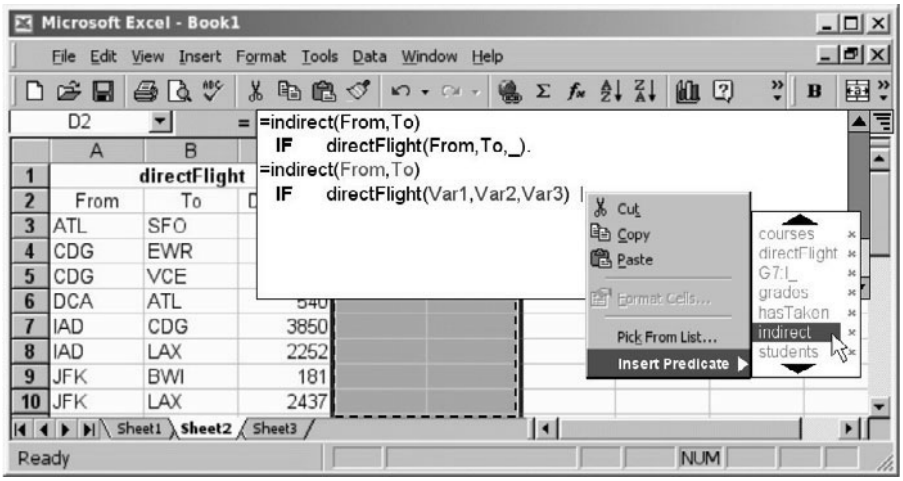


Figure 1 Mouse-Assisted Clausal Definition

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

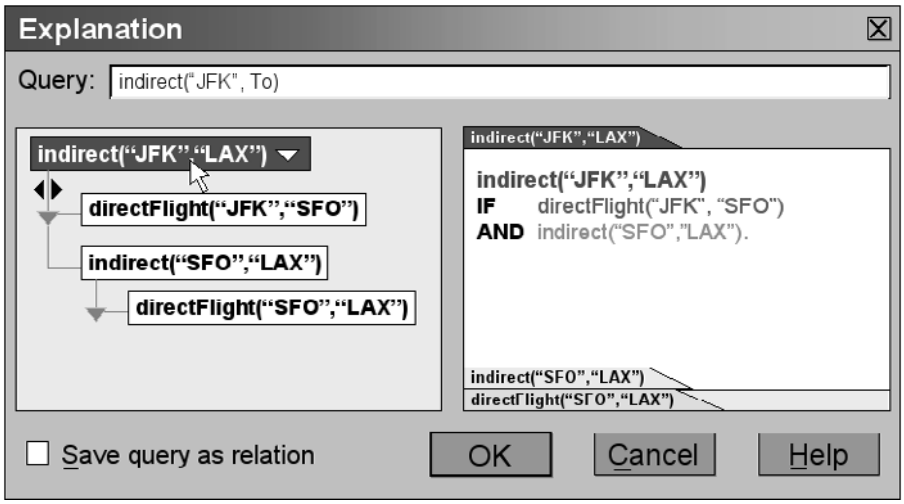


Figure 2 Deductive Explanation Dialog

The most immediate way to create logical definitions is to enter it into the formula input textbox, just as for scalar formulas. Because clausal rules (or SQL statements) tend to be more complex than a typical arithmetic formulas, we have enhanced this capability by effectively making available a syntactic editor that highlights keywords and provides visual feedback for relations and their columns, autoformats complex definitions, possibly over several lines, and monitors what the user writes in real time, recognizing potential errors and suggesting corrections. A second way to define a derived relation is to use the mouse to drag and drop predicates in a clause, or select them from dedicated menus. An example is given in Figure 1. Finally, the formula insertion wizard, although seldom used in practice, has been extended to support the defining clauses. The traditional cut-and-paste mechanism for propagating formulas, a favorite among users, has been extended to work seamlessly over logical formulas (and by extension over array formulas, whose support is rather deficient in commercial spreadsheet applications).

Core functionalities are invoked as in the Traditional Spreadsheet. In particular, evaluation is automatically performed when a file is loaded. Recalculation is automatic by default; but it can be fine tuned using menu dialogs, and forced by invoking dedicated commands. Explanation can be invoked by choosing a menu item, which brings up the dialog in Figure 2, which inquires about how outbound itineraries from 'JFK' have been obtained. The explanation mechanism allows the user to cycle through all possible answers to a query (here it examines itineraries between 'JFK' and 'LAX') and all the possible ways they can be obtained. For each, the left pane displays a directory-like structure that outlines the sequence of steps taken, that is, all the records it depends on. Because such explanations can be rather large, the user can minimize parts that are not deemed interesting, and incrementally expand them later. When any record is selected, the right pane shows the instantiated clause that produced it.

Traditional Spreadsheets come with a number of tools aimed at visualizing raw data in an intuitive manner, hence making them more readily usable, augmenting productivity and improving the user's experience (or his/her customers'). Many of these tools target tabular data, allowing to summarize them as charts or graphs, and to build complex what-if analysis scenarios with a few mouse clicks. The deductive extension opens the doors to new opportunities. For example, a relation such as directFlight can conveniently be displayed as a graph with airports as nodes and connections as edges possibly labeled with distance or other information. Overlaying other relations provides an immediate visual intuition that is not easily achievable in tabular form. We call this type of rendering a *connection graph*. Another productivity tools we have been investigating in the context of the Deductive Spreadsheet is a flow graph generator (Cervesato, 2005).

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

5 Preliminary evaluation and user feedback

We have conducted a small-scale and very preliminary usability experiment on potential users of the Deductive Spreadsheet. We interviewed eight current users of Traditional Spreadsheet applications, principally Microsoft Excel, with diverse backgrounds and sophistication levels. We explained the general idea underlying the proposed extension, showed them screenshots for a simulated example of the Deductive Spreadsheet and noted their reactions.

Altogether, this experiment showed that our target audience, namely intermediate and advanced users, could easily grasp the potential of the Deductive Spreadsheet. Indeed, they suggested several uses for it in their current activities. They commented favorably on the user interface, especially the explanation facility and the possibility of displaying tables as a connection graph. To our surprise, also beginners, which were not a target category, showed interest in the basic relational functionalities, although they had difficulties with some of more advanced concepts such as recursion. Beginners took issue with the wording of menu items and other interface text, and demanded friendlier approaches to building simple clausal definitions so that they could perform basic manipulations of tabular data in unintimidating ways. In essence, beginners pressed for being included as the beneficiaries of this technology.

Tested users across the board were very positive about the integration of persistent relational query facilities in a spreadsheet, a key aspect of the Deductive Spreadsheet. They were clearly interested in the possibility of using spreadsheets as small homegrown databases, without the steep learning curve of typical database management systems. The idea of tabular views of data where changes are automatically propagated seemed particularly appealing to them.

The feedback from volunteers in our target audience, advanced and intermediate users, suggests that we may have achieved our cognitive objectives. We also see the keen interest of beginners for what the core technology of the Deductive Spreadsheet can deliver as an unexpected niche. If rigorous testing confirms these preliminary results, we intend to concentrate future efforts to making available intuitive tools to take advantage of basic, but useful, aspects of our solution. We also plan to build on the numerous improvements suggested by the advanced and especially intermediate users.

6 Toward an implementation

At the time of writing, we are in the early phase of an implementation of the Deductive Spreadsheet into a prototype that we call NEXCEL. When completed, this system will embed all the primary operational functionalities discussed in Section 3, in particular evaluation, update and explanation over the entire linguistic extension, and a majority of interface functionalities previewed in Section 4. This implementation is intended as a testing ground for our design, and is principally aimed at assessing two aspects of this proposal: performance and usability. Abstract modeling (Cervesato, 2005) anticipates that, in practice, typical clausal definition will be evaluated in a time comparable to traditional formulas (and therefore have no tangible cost for the user), although complexity analysis indicates the possibility of a polynomial degradation in the worst case. An actual prototype will allow us to assess the performance for a wide range of usage, and also to experiment with optimizations. On the usability front, this prototype will permit actual user testing and experimentation with different ways of making functionalities available to users. Ultimately, it will prepare the stage for an actual implementation of the Deductive Spreadsheet, either an efficient module for an existing spreadsheet application (either Microsoft Excel or some other commercial product) or as an independent application.

The NEXCEL prototype consists of two components, as sketched in Figure 3: an add-on module to Microsoft Excel 2003 and an inference engine. Our implementation relies on an off-the-shelf Microsoft Excel 2003 executable for all traditional functionalities as well as most user interface operations. The add-on module, written in Visual Basic for Applications, acts as a bridge between Excel and the inference engine: it recognizes user interactions intended to access extended

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

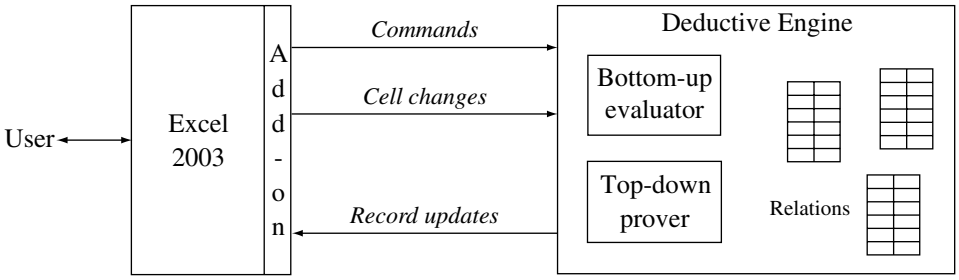


Figure 3 Structure of the NEXCEL Prototype

functionalities and dispatches them as needed to the inference engine. When loading a deductive spreadsheet from file, it will communicate the contents of all regions used relationally, as well as any defining clauses it finds, and install the evaluated records in the appropriate cells as soon as it receives them from the inference engine (this process will often be conducted over several iterations). Similarly, it will relay any change affecting the logical fragment of the spreadsheet and update deduced values as reported by the deductive engine. Finally, it will pass on the explanation requests and display their result as described in Section 4. It will directly handle the most basic requests only, such as some changes in the geometry of a relation.

The inference engine implements all the functionalities described in Section 3, evaluation, update and explanation, and maintains appropriate data structures for these purpose. These currently include an evaluated copy of every relation referenced or defined in the Deductive Spreadsheet, and of course the clauses defining the latter. They also include graphs tracking various types of dependencies between predicates, in particular their call graph, which includes stratification information needed during evaluation. Our current prototype is being written in the functional language Haskell, which combines reasonable efficiency and rapid prototyping.

7 Conclusion

In this paper, we have outlined a design for an extension of the Traditional Spreadsheet that supports powerful forms of symbolic reasoning while maintaining the usability of its user interface, as embodied in successful commercial products. This design is currently being implemented into the NEXCEL prototype.

Acknowledgements

We are grateful to numerous individuals for their useful comments on the early ideas underlying this work. Among them, special acknowledgments go to Burkhard Freitag, Georg Gottlob, David Gunning, Mike Kassoff, Frank Pfenning, Alberto Pravato, Alessandro Roncato and David Walker, as well as the participants of the WOLS'05 Workshop on Logical Spreadsheets. We are especially indebted to the users who volunteered some of their time to provide us with valuable comments on functionalities and user interface design.

References

Blackwell, A. 2002 First steps in programming: a rationale for attention investment models. In *Conference on Human-Centric Computing Languages and Environments*. IEEE Computer Society, pp. 2–10.

Boehm, B. W., Abts, C., Brown, A., Chulani, S., Horowitz, E., Madachy, R., Reifer, D., Clark, B. K. and Steece, B. 2000 *Software Cost Estimation with COCOMO II*, Upper Saddle River: Prentice Hall.

Ceri, S., Gottlob, G. and Tanca, L. 1990 *Logic Programming and Databases*, Springer Verlag.

Cervesato, I. 2005 The deductive spreadsheet. Technical Report DS05-02, Deductive Solutions. Berlin, Heidelberg, New York.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press

Cheng, M. H. M., van Emden, M. H. and Lee, J. H.-M. 1988 Tables as a user interface for logic programs, *in* 'Fifth Generation Computer Systems', pp. 784–791.

Colomb, R. M. 1998 *Deductive Databases and their Applications*, London: Taylor & Francis.

Green, T. and Petre, M. 1996 Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages and Computing* **7**, 131–174.

Gupta, G. and Akhter, S. 2000 Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Proceedings of the Second International Workshop Practical Aspects of Declarative Languages*. Boston, MA: Springer Verlag LNCS 1753, pp. 308–323.

Kriwaczek, F. 1982 Some applications of Prolog to decision support systems. Master's thesis, Imperial College, London, UK.

Kriwaczek, F. 1988 Logicalc: a Prolog spreadsheet, *Machine intelligence*.

Lloyd, J. W. 1987 *Foundations of Logic Programming*, 2nd extended edn. New York: Springer-Verlag.

Panko, R. R. 1998 What we know about spreadsheet errors. *Journal of End User Computing (Special issue on Scaling Up End User Development)* **10**(2), 15–21. Available at `http://panko.cba.hawaii.edu/ssr/Mypapers/whatknow.htm`.

Peyton Jones, S., Blackwell, A. and Burnett, M. 2003 A user-centred approach to functions in Excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, Uppsala, Sweden: ACM Press, pp. 165–176.

Power, D. 10/04/2003 A brief history of spreadsheets. Available at `http://www.dssresources.com/history/sshistory.html`. DSSResources.Com.

Smith, D. E., Genesereth, M. R. and Ginsberg, M. L. 1986 Controlling recursive inference, *Artificial Intelligence* **30**(3), 343–389.

Spenke, M. and Beilken, C. 1989 A spreadsheet interface for logic programming. In *CHI '89: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM Press, pp. 75–80.

van Emden, M. H., Ohki, M. and Takeuchi, A. 1986 Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing* **4**(3), 287–304.

Warren, D. S. 1998 Programming with tabling in XSB, In D. Gries and W. P. de Roever (eds.), Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET'98), Vol. 125 of *IFIP Conference Proceedings*, Shelter Island, NY: Chapman & Hall, pp. 5–6.

Workshop on Logical Spreadsheets 2005 Available at `http://wols05.stanford.edu`.

https://doi.org/10.1017/S0269888907001142 Published online by Cambridge University Press