Tables as a User Interface for Logic Programs*

M.H.M. Cheng M.H. van Emden J.H.M. Lee

Abstract

Spreadsheets have introduced two advantages not typically available in user interfaces to logic programs: the exploratory use of a computer and a two-dimensional interface. In this paper we show that not only spreadsheets, but also tables (in the sense of relational databases) have these valuable features. We compare spreadsheets and tables, giving possibly the first clear distinction between the two and suggest a common generalization. We show that tables, as a user interface for logic programs, can be derived from a dataflow model of queries (which we call TuplePipes), which provides also the buffering needed when Prolog is interfaced with a relational database. We report on Tupilog, a prototype implementation of logic programming allowing four query modes, one of which is TuplePipes.

1 Introduction

Suppose you want to use a computer to construct a complex data object satisfying a large number of constraints, such as a schedule or a budget. There are two ways of going about it — a planned way and an improvised way. The planned way has been established longest in computing: make sure you have available in advance all constraints and process them all in one go. If you forgot something, then you will discover this when you have reconstructed the solution from the output, and you can have another try. This cycle has been speeded up enormously with the transition from batch processing to the use of interactive terminals, but is still slow compared to the other, more recently developed alternative, which we now describe.

Since the advent of personal computers and spreadsheet software you have an alternative that we call the improvised way. Here you don't need to know in advance all constraints; you can discover them by looking at an unsatisfactory solution which itself suggests the lacking constraints. Spreadsheets allow you to proceed in an exploratory fashion, trying this, trying that, doodling with your data until you hit upon a satisfactory solution.

Let us now consider how logic programming caters to the planned and improvised ways of using a computer. Logic programming can be summarized as a way of using a computer where the relation between input and output is defined in predicate logic and where a suitable interpreter uses this definition together with given input data to construct the corresponding output. Although this principle allows both planned and improvised modes, common Prolog implementations have all followed the first option, requiring the user to irrevocably terminate the query before showing the answer substitution. But, as was shown in [7], the spreadsheet user interface allows logic programming to be used in improvised mode.

In this paper we show that the potential of logic programming for the improvised, exploratory mode of computer use is not exhausted by the spreadsheet interface. In addition, *tables* are a medium suitable

*Dept. of Computer Science, University of Victoria Technical Report DCS-97-IR. Also published in Proc. Int. Conf. on Fifth Generation Computer Systems, ICOT, 1988.

name	test 1	test 2	test 3	s_{avg}
ayre	69	47	49	55
bell	74	76	84	78
coe	82	82	85	83
dare	58	56	90	68
eames	82	72	71	75
fixx	44	56	41	47
gore	81	59	91	77
t_{avg}	70	64	73	69

Table 1: A display which can be a spreadsheet or a table.

for doodling with data. We show that tables also provide logic programming with a congenial user interface for an exploratory mode of computer use.

Here is a brief overview of this paper. To start with, it is important to make clear the difference between spreadsheets and tables; this is the topic of the next section. The close connections between tables, relational databases and logic programming are the topic of section 3. In section 4 we show how incremental queries, the mechanism used in [6] to make exploratory computer use possible in the framework of logic programming, can be adapted to tables as well as to spreadsheets. As tables give a relational model of data, it is important to connect the common operations on relations to the goals of logic programming; this we do in section 5. Finally, we devote a section to a short description of Tupilog, an implementation of the TuplePipes concept, which has allowed us to experiment with the ideas developed in this paper.

2 Stacks, spreadsheets and tables

Spreadsheets and tables are closely related, yet have significant conceptual differences. Spreadsheets are isotropic: the horizontal and vertical directions play the same role. In a table, the rows and columns are essentially different. This is because a table represents a relation in the sense of the relational data model: each row is a tuple of the relation represented by the table and each column is an attribute.

Consider as an example the display in Table 1, showing the result of seven students in three tests, together with averages per student and per test.

If the display is a table, then it represents a relation. In the mathematical sense, an n-ary relation is a set of n-tuples. These tuples are listed in the rows of the table, except for the top and bottom rows, which play a special role. The relation in the example is a five-place relation, consisting of seven five-tuples. Thus we see that in the table the rows and columns play different roles. Also, the averages for the students (`s-avg`) are part of the relation, but those for the tests (`t-avg`) are not. If the display in Table 1 is regarded as a spreadsheet, we do not make such distinctions: rows and columns have the same status.

Spreadsheets got to computers before tables did, exploding into the vacuum of the initially empty niche for exploratory computer use. As a result there are many spreadsheet programs available and only a few based on tables. We have used WATFILE, a table-based data manipulation system originated by J.W. Graham at the University of Waterloo and subsequently developed at the Computer Systems Group in Waterloo, mainly by a group under Terry Wilkinson [8].

For many applications the relational data model is the natural one and these are better served by tables than by spreadsheets. Yet, because of the predominant position of spreadsheet programs, such

applications are often forced into the, for them, less appropriate spreadsheet model. Because logic programming is compatible with the relational data model, it should come as no surprise that tables are a natural user interface for logic programs. The next section will establish the necessary connections between tables, logic programs, and relational data bases.

But, in [7] it was shown that spreadsheets are also a natural user interface for logic programs. To resolve this apparent contradiction, we need some additional explanation. In a table, the relation relates the various attributes of the tuples. In a spreadsheet, there are constraints between the cells. Note that, though "constraint" is often used in the context of spreadsheets, it also means "relation". Thus, one way of distinguishing spreadsheets from tables is to say that in a table the constraints only go in the horizontal direction, whereas in a spreadsheet they go either way.

Imagine a table with very long tuples, much too long to fit on a screen. It may well be that the constraints within such a tuple can be visualized better when its elements are displayed in a two-dimensional array. That is, such a big tuple is a little spreadsheet by itself. According to this view, a table becomes a stack of spreadsheets. Such an object amounts to a *common generalization of spreadsheets and tables*, which we shall refer to as a *stack*.

It relates the spreadsheet interface for logic programs presented in [7] to the table interface to be developed in the present paper. Stacks may also serve as a basis for improved ad-hoc spreadsheet software: often spreadsheets become very large, not because the constraints extend over large distances, but because the large spreadsheets consist of many small ones next to each other. In such a situation a stack is more appropriate than either a table or a spreadsheet.

3 Tables, logic programs, and relational databases

A logic program can be regarded as a virtual relational database: one in which not all tuples are stored explicitly, but are typically generated on demand by means of the rules stored in the program. Goal statements to logic programs play the role of database queries, with Query-By-Example as the query language in spirit closest to logic programming.

This compatibility of logic programming with the relational data model suggests that Prolog be interfaced to a relational database serving as a back end. Then the relations in the database can be presented to the Prolog user as built-in predicates, having the same status as the ones for arithmetic have. But Prolog's backtracking control causes it to request the tuples from the database one at a time so that such a scheme would not be usable without buffering.

One could of course regard such buffering as a low-level implementation detail to be kept out of sight in a discussion on user interfaces. However, a query can be viewed as a dataflow network, in such a way that the network is conceptually helpful to the user, and can be regarded as an implementation of the buffering required when interfacing Prolog to a relational database.

In the dataflow network representing a query, the goals are nodes and are connected in series by channels transmitting partial answer substitutions. All goals to the left of a certain channel form a query in their own right. If this partial query succeeds (and it must, if the entire query is to succeed), then there are answer substitutions, and these are transmitted through that channel.

Let us illustrate the dataflow model with an example based on the data in Table 1. When we view these data as a relational table, then the relation has five arguments: `name`, `testl`, `test2`, `test3`, and `s-avg`. But, as is often handier in practice, we can single out one attribute, in this case the name, as primary key and use binary relations to relate each of the other attributes to it. We call these binary relations `tl`, relating the name to `testl`; `t2` and `t3` are similarly defined. With these relations we can build up a table as in Table 1 step by step by means of a query with several goals; see Figure 1 for an example.

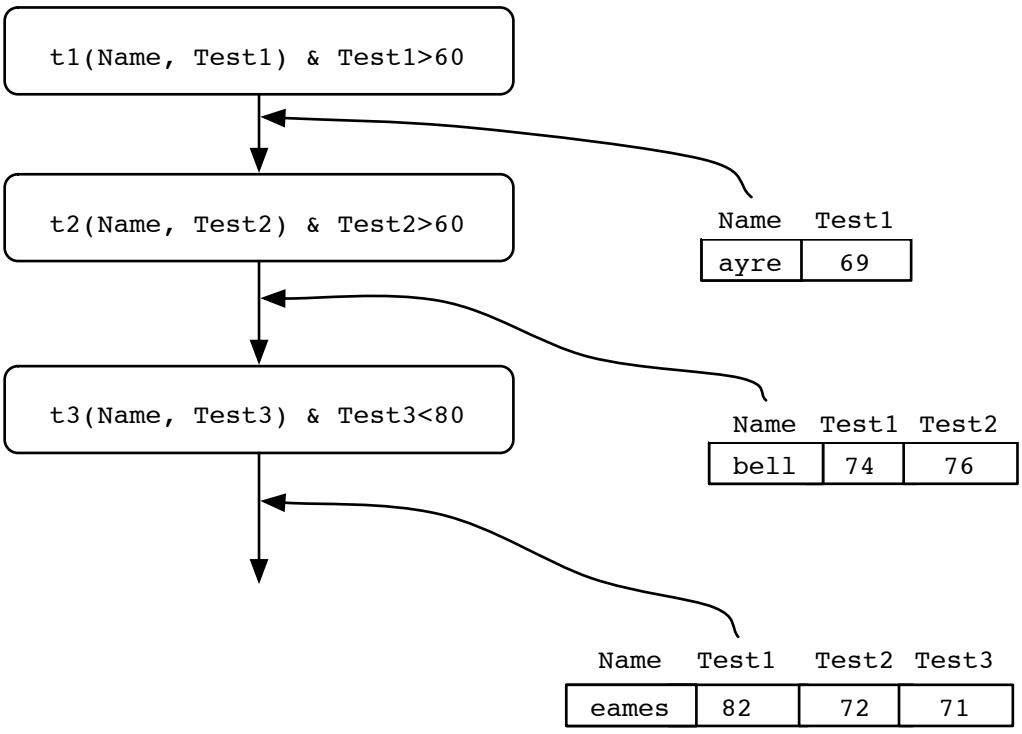

Figure 1: A query and its dataflow network; a single tuple in each pipe.

Each answer substitution is a tuple of variables, each possibly with a binding. The dataflow channel familiar to most people is the “pipe” of UNIX, so that we see tuples flowing through pipes. Hence the name ”tuplepipes” for our prototype implementation described in a later section.

In a dataflow network, the nodes can be scheduled in such a way that

- the pipes are kept as empty as possible. Then there is at most one tuple in each pipe, which corresponds to the way Prolog evaluates queries. This regime is not suitable for interfacing with a database back end as there is no buffering and tuples are extracted from the database one at a time.
- the pipes are allowed to hold a fairly long sequence of tuples. This sequence is then the buffer required for an economical interface to a database back end for logic programs. We advisedly do not say “Prolog”, as this does not correspond to the way Prolog evaluates queries.

But, however nodes are scheduled, the dataflow model itself already determines the sequence of tuples that flow through each pipe at one time or another. It is this sequence that we are interested in: it is the sequence of all answers to the query formed by the goals upstream from the pipe. *This sequence of tuples is a table*. Thus the query can be viewed as the first goal generating a table and each next goal transforming the table determined by its input pipe to the table determined by its output pipe, which is the input pipe of the next goal. (See Figure 2.)

4 Incremental queries for a tabular interface

In conventional Prolog, the user can only see an answer substitution when the query is irrevocably terminated. Should the user have forgotten about a constraint, then there is no alternative but to start over.

Although this may be convenient from the implementer's point of view, it is certainly not so for the user who wants to use Prolog in an exploratory mode to construct a complex data object, such as a budget or a schedule, subject to many constraints. Then one typically does not know explicitly in advance all constraints the output should satisfy; when confronted with a candidate output, the user may find it unsatisfactory and may be reminded of a constraint that was omitted. An implementation where the query may be continued after display of a preliminary answer substitution, does allow such exploratory computer use. We call such a facility an *incremental query interface*. It was first proposed in [6], applied to spreadsheets in [7], and further developed in [5].

Semantically, incremental queries present no difficulties. They are based on the observation that, for a query to succeed, every initial segment of it has to succeed as well. Each of these is a query in its own right, and has an answer substitution if it succeeds. The only difference between a conventional query interface and an incremental one is that the former has a single command for the distinct functions of displaying the answer substitution so far and for terminating the query; in the latter these functions have separate commands.

In an incremental query, the user can view a goal as an operator for changing the previous answer substitution to the next. When there are many variables in an answer substitution arranged in a two-dimensional matrix, then we can think of them as the cells of a spreadsheet. The goals then specify relations between these cells. For further information we refer to [7].

Let us now consider the consequences of incremental queries for the dataflow model. The usual batch type queries of Prolog correspond to complete execution, without possibility for interaction, of the entire network: the first and only thing the user sees is the rightmost pipe. If the query is incremental, then the user can see the pipes' contents, as they are filled from left to right. In other words, the *user can view each goal as an operator that changes the previous pipe into the next one*. This is the key concept of both spreadsheets and TuplePipes as user interfaces for logic programs. In the spreadsheet case, it is a single tuple in spreadsheet form that is transformed by a goal. In the TuplePipes case, it is the sequence of all answer tuples, that is, a table that is transformed by a goal. Thus, TuplePipes are intimately interactive in a way similar to that of spreadsheet programs or of WATFILE: the user can decide on the basis of the current state of the display what action to take next; i.e. what goal to enter next to effect the desired operation on the current table.

5 Relational operations

In TuplePipes, every additional, incrementally processed, goal is an operator acting on a table, yielding a new table. Special cases of these operators correspond to standard relational operations, such as join, selection, and projection. But in general, the goal atoms of logic programming provide a flexible and powerful repertoire corresponding to infinitely many relational operations. Although we believe that the standard operations are of little practical importance, it is useful to consider them once and to satisfy oneself that they are available in the context of logic programming. We give examples to illustrate how joins, selections, and projections can be done in logic programming. The examples are based on the data introduced in Table 1, assuming that the available relations are the binary `t1`, `t2`, and `t3` used in Figure 1 and Figure 2.

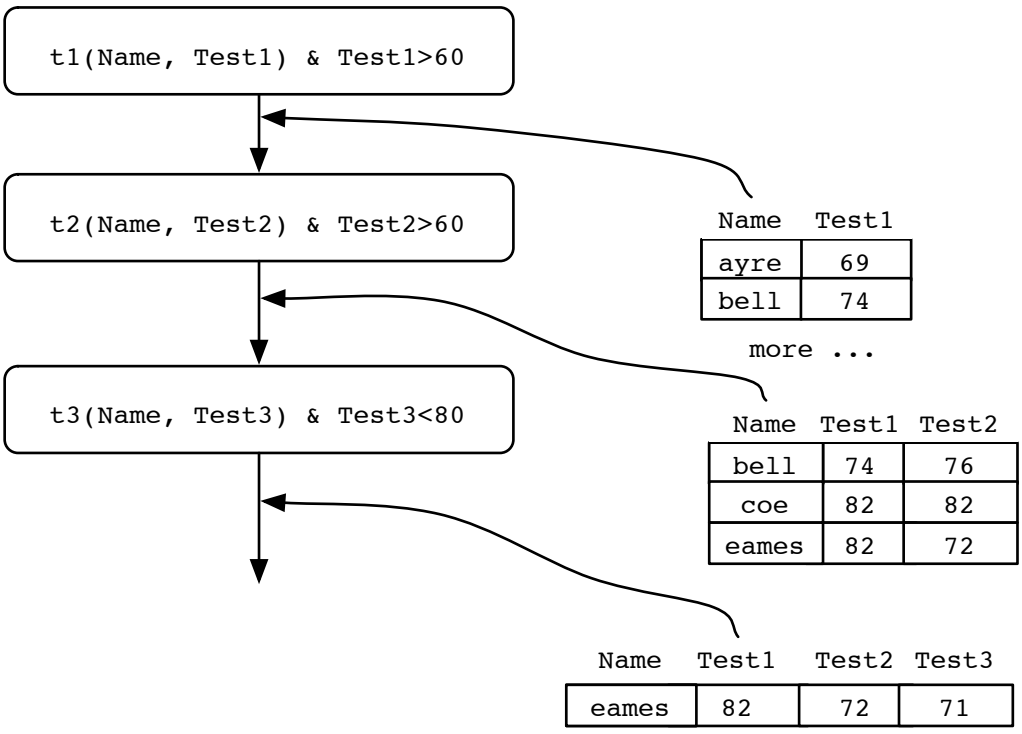

Figure 2: A query and its dataflow network; all tuples in each pipe; each pipe a table.

Consider the query

```
? tl(Name,Testl) & t2(Name,Test2);
```

View this query as a dataflow network consisting of a node for each of the goals with a pipe between them. Then the sequence of the tuples in this pipe constitutes the input table for the second goal and this goal performs a *join* on this table. In the query

```
? t1(Name,Testl) & Testl>60;
```

the second goal performs a *selection* on its input table. A goal, like this one, that does not introduce a new variable is restricted to performing selections. But a goal, like in the previous example, that does introduce a new variable, can perform selection as well as join. The data in Table 1 are untypical in that the second goal in the first example performs a join only, without any selection.

In most logic programming systems, projections can only be done by introducing a predicate and defining it by means of an additional rule; it cannot be done in a query only. For example, in

```
new(Name,Avg) <-
        tl(Name,Test1) &
        t2(Name,Test2) &
        t3(Name,Test3) &
        avg(Testl,Test2,Test3,Avg);
```

```
? new(Name,Avg);
```

the output of the query considered as dataflow network is a table which is the projection of the table in Table 1 onto the first and last columns.

In conventional Prolog, the user has to be in planned mode, so it does not matter much that projections have to be prepared in advance by adding the required rules. In TuplePipes, however, where the user is improvising a long incremental query it is a serious handicap to have to interrupt a query every time one has to do a projection. We have therefore incorporated a facility to allow the user to perform the projection, and to name the result, as part of the query. We do this by entering a right arrow followed by a new predicate symbol and those arguments to be included in the projection. For example, again with the data of the previous examples,

```
? t1(Name,Testl) &
  t2(Name,Test2) &
  t3(Name,Test3) &
```

We look at the table; find it contains unsuitable tuples.

```
Test1>60 &
```

Another look shows that the table is still not what we want.

```
Test2>70 &
```

All unwanted tuples have been removed; we take the averages for the remaining students.

```
avg(Test1,Test2,Test3,Avg) &
```

This table is the one we want, worthy of being enshrined in a relation of its own. But we are no longer interested in individual test results, so we project. The resulting table is named `new`.

```
new(Name,Avg) &
```

Oops! We forgot to exclude averages that are too high. Note that as a result of the projection we are now operating on a table with `Name` and `Avg` as only columns. We continue with a goal that selects.

```
 Avg < 80 &
```

And so on . . . Note that the projection and naming operation fit well in the logic programming framework, because the query up to and including the operation reads just like the rule that is normally needed to prepare for the same operation. The only difference is that now the conclusion of the rule is to the right of the conditions, and the direction of the arrow is changed accordingly.

Note that the expression

```
-> new(Name,Avg)
```

is a component of the query in the same sense that a goal is: they are both operations on tables. After

```
-> new(Name,Avg)
```

the TuplePipes query may continue with the table consisting only of the columns Name and Avg.

6 Tupilog, the implementation

The TuplePipes project reported here is part of a continuing effort to make the full power of video monitors available for exploratory computer use by means of a logic programming language. The first effort in this direction was the project reported in [7], done at ICOT. Its contributions were the first implementation of incremental queries, and of a spreadsheet interface based on answer substitutions. See [3] for an alternative approach based on assertions. This project allowed a first glimpse of the possibilities of exploratory programming in logic. However, the hardware (a VT100 terminal connected to a DEC10 computer) made it difficult to go beyond the glass teletype kind of user interface.

In the TuplePipes project we had available more powerful equipment (at least from the user interface point of view): a personal computer with a mouse. The additional novelty was the requirement to demonstrate tables rather than spreadsheets.

The implemented system is basically Prolog with four different query modes which can be classified in single-answer modes and all-answer modes. Each of the classes is subdivided into two variants: one that is not incremental and one that is. Thus we have four possibilities in all:

- a non-incremental, single-answer mode, which corresponds to the only mode provided by most Prolog implementations,
- a non-incremental, all-answer mode, as provided by Turbo Prolog,
- an incremental, single-answer mode corresponding to the one of the system reported in [7],
- the main component of this project, the only one deserving the name TuplePipes: the incremental, all-answer mode.

Thus we should distinguish between TuplePipes, the name of the dataflow model giving rise to tables, and the implemented system, which we call Tupilog, and which has TuplePipes as one of its four modes of interaction.

It is this last mode that exploits best the many windows provided by our computer system. Initially there is one window, which we call the meta window in which you can enter, in Prolog query format, a goal specifying the initiation of any of the four modes. Let us suppose it is the incremental, all-answer mode. As a result an initially empty new window appears, in which the goals of the incremental query will be displayed. As soon as the first increment is entered in that window, a new window appears containing the table resulting from the first goal. Every next goal entered in the query window transforms the table into a new one, which then takes its place in the table window. Typically, the table window does not hold all of the table. Scrolling allows you to select the visible part.

You can create other modes, each with its own set of windows. You can use the mouse to rearrange the windows, to change their dimensions, and to bring selected ones to the foreground.

The system consists of four meta interpreters, one for each mode, implemented in about 250 clauses (about 1000 lines, including comments) of ALS Prolog [2]. We use meta-level constants to name object-level variables. At the meta level we have implemented operations on table columns, such as summing, averaging, and sorting. These operations are logically intractable within the object level, but provide no difficulty when there is a clear distinction between object and meta levels. Also, this distinction facilitates the implementation of the projection operation described in the previous section.

Of the four meta interpreters, the only one we need to say a bit more about in this paper is the one for TuplePipes. We list here its top level, sufficient to specify the computational model.

```
% nodes(GoalList,InPipe,outPipe) :
% InPipe (OutPipe) is the stream of tuples
```

```
% through the input (output) pipe of the
% dataflow nodes corresponding to GoalList.

nodes([],Pipe,Pipe).

nodes([Goal|Goals],InPipe,OutPipe) :-
  node(Goal,InPipe,Pipe),
  nodes(Goals,Pipe,OutPipe).

% node(Goal,InPipe,OutPipe) :
% InPipe (OutPipe) is the stream of tuples
% through the input (output) pipe of the
% dataflow node corresponding to Goal.
%
% allAnswers(Goal,Tuple,Pipe) :
% Pipe is the sequence of answers to Goal
% with Tuple applied to it as substitution.

node(_,[],[]).
node(Goal,[Tuple|Tuples],OutPipe) :-
  allAnswers(Goal,Tuple,OutPipe1),
  node(Goal,Tuples,OutPipe2),
  append(OutPipe1,OutPipe2,OutPipe).
```

Most input and output, especially with the mouse, was implemented directly in about 2000 lines of Microsoft C (including comments), bypassing the I/O of ALS Prolog, but using its C interface routines. To facilitate the manipulation of the windows and the interface to the mouse, the "C Utility Library" (supplied by Essential Software, Inc.) was used. It all runs on an IBM XT/286 under PC/DOS.

It is important to distinguish TuplePipes, the conceptual user interface, from Tupilog, the implementation. The implementation is described here without going into details because it is useful only as prototype allowing us to demonstrate the intended user interface.

For a more serious implementation, one thinks of course of the And-parallel languages such as Parlog, Concurrent Prolog, and GHC, if only because of the large amount of effort spent on their implementation. These languages are relevant because they allow one to implement dataflow networks based on logic. But these networks are a very special case of what And-parallel languages can do in general.

TuplePipes are themselves a special case of dataflow networks. Yet, as we have argued, they support a large part of what many users want to do with their computer. This combination of conditions strongly suggests not to rely on the general power of And-parallel languages, but to use a special-pupose implementation such as Tupilog with a computational model inspired on the processor for logic programs in [1] and in [4].

7 Conclusions

It is interesting to see how far software lags behind hardware. A case in point is the transition from teletypes to video monitors as terminals for computers. The monitors allow a superior, two-dimensional interface. Yet, although teletypes have long disappeared, much software is still line-oriented, using the monitor as a "glass teletype".

Strikingly, this not only holds for languages like Pascal and C, but also for interactive languages like Prolog and Lisp. And it not only holds for old-fashioned terminals like the VT100, but also for workstations. Their innovation has been to provide multiple windows. But each of these contains . . . a little glass teletype.

We feel that logic programming has learned from what's happening in the outside world, where the first experiments in exploratory computer use were conducted with spreadsheets, which also pioneered a step away from the glass teletype. On the other hand, logic programming has contributed to the outside world, showing that spreadsheets and table-based software can be interfaces to logic specifications expressing directly the user's requirements, making it easier to verify that all those numbers that come pouring out are indeed a solution of the problem at hand and not of something else.

Modeling spreadsheets and tables in logic helps to bring out their differences and similarities. Our analysis, and our notion of their common generalization, the stack, may be new, and is a consequence of the need to use them as a two-dimensional interface for logic programs.

8 Acknowledgements

We gratefully acknowledge Doug Teeple for his enthusiastic encouragement and the Toronto Laboratory of IBM Canada for their generous support of this project.

References

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge,Massachusetts, 1985.

[2] ALS. *ALS Prolog Technical Reference Manual, Professional Version 1.0*. Aplied Logic Systems, Inc., P.O. Box 90, University Station, Syracuse, NY 13210, 1986.

[3] F. Kriwaczek. Logicalc — a Prolog spreadsheet. In D. Michie and J. Hayes, editors, *Machine Intelligence 11*, 1987.

[4] G. Lindstrom and P. Panangaden. Stream-bases execution of logic programs. In *International Symposium on Logic Programming*. Computer Society Press, 1984.

[5] M. Ohki, A. Takeuchi, and K. Furukawa. A framework for interactive problem-solving based on interactive query revision. In E. Wada, editor, *Proceedings of the 5th Conference on Logic Programming*, pages 137–146. Springer-Verlag LNCS 264, 1986.

[6] M.H. van Emden. Logic as an interaction language. In *Proc. 5th Conf. Canadian Soc. for Computational Studies in Intelligence*, pages 126–128, 1984.

[7] M.H. van Emden, M. Ohki, and A. Takeuchi. Spreadsheets with incremental queries as a user interface for logic programs. *New Generation Computing*, 4:287–304, 1986.

[8] T. Wilkinson. *WATFILE/Plus Data Manipulation System*. WATCOM Publications Ltd, Waterloo, Ontario, Canada, 1985.