TR-144

Spreadsheets with Incremental Queries as a User Interface for Logic Programming

by
M. H. van Emden(Univ. of Waterloo),
and
M. Ohki, A. Takeuchi(ICOT)

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Spreadsheets with Incremental Queries as a User Interface for Logic Programming

M.H. van Emden
University of Waterloo
Waterloo, Ontario, Canada

and

M. Ohki A. Takeuchi
Institute for New Generation Computer Technology
Tokyo, Japan

Abstract

We believe that currently marketed programs leave unexploited much of the potential of the spreadsheet interface. The purpose of our work is to obtain suggestions for wider application of this interface by showing how to obtain its main features as a subset of logic programming. Our work is based on two observations. The first is that spreadsheets would already be a useful enhancement to interactive languages such as APL and Basic. Although Prolog is also an interactive language, this interface cannot be used in the same direct way. Hence our second observation: the usual query mechanism of Prolog does not provide the kind of interaction this application requires. But it can be provided by the Incremental Query, a new query mechanism for Prolog. The two observations together yield the spreadsheet as a display of the state of the substitution of an incremental query in Prolog. Recalculation of dependent cells is achieved by automatic modification of the query in response to a new increment that would make it unsolvable without the modification.

1 Introduction

We belong to a growing group of scientists working toward a situation where computers are universally used as tools to empower human minds. Not just the minds of programmers or clever non-programmers, but of any non-retarded, toilet-trained human. Of course, natural language will play a role in achieving this situation, but we hardly have an idea of the work that needs to be done. Given our ignorance in this area, is our goal perhaps over-ambitious?

We believe not. It is clear that natural language, though indeed natural, is not the preferred vehicle of thought in all areas. Centuries ago, problems, even numerical ones, were exclusively stated and solved in natural language. Renaissance mathematicians discovered that algebra is a superior language for a large class of such problems. About one century ago Frege found that problems of non-numerical reasoning can often be solved easier in the symbolic calculus he invented.

Even older, perhaps, is the saying *A picture is worth a thousand words*. Note that these are words of natural language. Recently, the iconic user interface of the Apple Macintosh computer has proved its effectiveness not only with the recently toilet-trained, but also with their parents and grandparents. Electronic spreadsheets, perhaps not quite iconic, but certainly not natural language, have taken the world of business by storm. Together with word-processing programs, they have proved that universal accessibility of computers has been achieved already.

But of course this desirable degree of accessibility exists only in a very narrow area. We believe that logic programming has the potential of dramatically enlarging that area. A problem that remains to be addressed is to construct a universally accessible user interface for logic programs. Approaches via natural language are being pursued, and these are promising. In this paper, however, we are concerned with the unexploited potential of the spreadsheet interface.

We explain first (section 2) in what sense spreadsheets are more widely applicable than in VISICALC, MULTIPLAN, LOTUS 1-2-3, SUPERCALC, etc. (we will refer to such programs as TYPICALC). We find that this wider applicability includes Prolog, though not via the queries typically used with Prolog. We use incremental queries, a modified type of query proposed in [1]. They are Prolog queries which are incrementally entered by the user. These are briefly reviewed in section 3, which also includes a front end for processing incremental queries, something that was missing from [1].

When the goals of an incremental query are restricted to equations, we almost have TYPICALC as a subset of Prolog. What remains to be added is the facility for changing cells and recalculating cell contents dependent on the ones changed. We discuss a prototype Prolog program for this in section 4. It depends on automatically modifying sets of equations that have become unsolvable. In section 5 we argue that a more general facility allowing the user to cancel arbitrary constraints is a powerful tool in interactive problem solving. A Prolog prototype for this facility is discussed as well.

2 How to generalize TYPICALC

In interactive languages such as APL or Basic, variables are typically created implicitly by executing an assignment containing a new variable name at the receiving end. Compare this with programs in an Algol-type language, where immediately after a declaration all the declared variables exist, but do not have a value. Furthermore, in the Algol-like languages as well as in APL or Basic, the value of a variable is only displayed as a result of an explicit output statement.

The method of APL or Basic is not the only way of obtaining the convenience of not having to declare variables. TYPICALC suggests another way: to have a standard, implicit declaration creating a fixed number of variables with fixed names. In TYPICALC the names identify cells in a two-dimensional matrix with one dimension specified by short, alphabetically ordered identifiers and the other by numbers. The advantage is that no explicit output statements are necessary: the matrix of values is constantly displayed, updated to the latest values. The disadvantage of non-mnemonic variable names is compensated by

a more effortless interaction.

Let us consider a variant of APL or Basic, where, as in TYPICALC, a fixed (though large number) of variables with standard, two-dimensional names always exits. Imagine that statements are restricted to assignments without any user-defined operations in the expression part. Let us call the *live part* of such a program the subset of assignment statements not superseded by later ones. An interesting consequence of such a restriction is that not only the state of the computation has an easily displayable matrix format, but also the live part of the program itself: instead of writing $<var>=<expression>$ we can write $<expression>$ in the cell with coordinates given by $<var>$.

Such an assignment-only program in matrix format can be extended by entering an expression into an empty cell or it can be modified by replacing an expression in an occupied cell. In the latter case one could adopt the convention that automatically all other expressions are recomputed which depend on the assignment just changed. Thus we see that the essential features of TYPICALC are modeled by assignment-only programs in a language such as APL or Basic equipped with an implicit standard declaration of variables. When interacting with such a program two matrix displays are relevant: one containing the program itself and the other containing the current values of the variables. In TYPICALC most of the screen is devoted to a display of the latter. In this display one cell may be selected. As a result of this selection the contents of the corresponding cell in the program matrix is also displayed.

In this way we have indicated that the matrix-display interface of TYPICALC can be generalized by adding to an interactive language like APL or Basic implicit, standard, matrix-shaped declarations. One should not hastily conclude that assignment-only programs are overly restricted just because the built-in operations of TYPICALC are those learned in primary school plus a few financial operations. Indeed the operations could include *lambda*, *apply*, the fixpoint operator Y, as well as APL operations. Such an enriched version of TYPICALC would provide a gradual transition for the universal audience of existing TYPICALC to the world of the expert programmer. We, however, have more ambitious extensions in mind: to provide a bridge also to relational databases and rule-based expert systems via logic programming.

3 A spreadsheet interface for logic programs

In this section we introduce the two basic ingredients of our spreadsheet interface for logic programs: incremental queries (section 3.1) and matrix displays for substitutions. We discuss in some detail (section 3.2) a prototype implementation of the former. We also built a prototype matrix display, which we only discuss in general terms (Section 3.3), as it is specific to the particular combination of display terminal (DEC VT131) and Prolog implementation (Edinburgh DEC20 Prolog).

3.1 Incremental queries

Up till now we have only mentioned imperative languages as candidates for a spreadsheet interface. But is not Prolog an equally good candidate? As we shall see, the

answer is *yes*, and *no*. It could be *yes* because spreadsheet cells can be used as variables for Prolog queries. Furthermore, we could consider the special case of logic programming where queries only have equality goals with at least one variable, say, v. The goal would then be represented by entering the other term of the equality goal in the cell of v.

The answer would be *no* if we would have to use Prolog queries in their conventional form: only after a query is complete, is any of its goals solved. The first opportunity a user gets for reacting to a computation is in the next query. But the variables in this query are distinct from those in the previous one, even if they have the same name. Thus, before the user starts on the next query, the value matrix has to be erased. The prized feature of TYPICALC, where a value matrix can be interactively built up, is incompatible with conventional queries of Prolog.

What is lacking in a typical Prolog implementation is a facility for users to enter a query in *increments*, each increment consisting of one or more goals. As each increment is entered, the system solves the query as far as entered and displays the answer substitution found so far. This is the idea of the incremental query, first introduced in [1]. It is based on the observation that if the query

$$?A1, ..., Am, B1, ..., Bn$$

has a solution, say, with answer substitution s, then so do

$$?A1, ..., Am \quad and \quad ?(B1, ..., Bn)s1$$

where $s1$ is the answer substitution to the first increment. If $s2$ is the answer substitution for the second increment $(B1, ..., Bn)$, then s is the composition of $s1$ and $s2$.

3.2 A program for incremental queries

First we develop a program in Edinburgh Prolog [2] for incremental queries. We do so in several steps, starting with a simple program not having all desired properties. We introduce these properties in several stages. In this way, the reader will understand why our final version is not simpler.

A sequence of goals may be solved under control of a user program by the following clauses:

```
solve([Goal|Goals]) :- call(Goal),solve(Goals).
solve([]).
```

This is of course unsatisfactory, not being incremental: the entire sequence of goals has to be specified in advance. Hence the following proposal:

```
iq :- read_query(Query),(Query=ok ; call(Query),iq).
```

Here the comma stands for conjunction; the semicolon for disjunction. Although this allows for repeated input of queries by the user, it fails to solve them as multiple increments of a single query. This failure has two causes:

(a) When a query increment fails, this version does not re-try an earlier increment to see if, with an alternative solution, the later increments would succeed.
(b) The same variable identifier occurring in different increments names different variables. It should denote the same.

Let us attend to these shortcomings one by one, considering (a) first. The reason why our last version could not re-try was that the earlier increments were not stored. We therefore introduce an argument containing the list of increments entered so far. We call it Stack because the increments are re-tried in reverse chronological order.

```
iq(Stack) :- read_query(Query),solve_query(Query,Stack).
solve_query(ok,_) :- !.
solve_query(Query,Stack) :- solve(Query,Stack).
solve(Query,Stack) :- call(Query), iq([Query|Stack]).
solve(Query,[Prev_Query|Stack]) :-
        solve(Prev_Query,Stack),call(Query).
```

However the above program is still incomplete since queries in Stack are changed by answer substitution. The problem will be solved together with problem (b). Our solution to the problem (b) is to have the program create an *environment*, a term containing as constants the identifiers for variables occurring in all increments up to a certain point in time. The environment associates a variable with each such identifier. When the term representing an environment is printed out, the variables appear as system-supplied identifiers (underscore followed by numeral in Edinburgh Prolog[2]). These are therefore different from the associated constants, which are printed as the same identifiers input by the user.

Stack has entries of the form Input+Env. Here Input is the query increment as entered by the user, that is, all variables appearing in the query are represented by constant identifiers. Env is the environment as it existed up to and including the query increment it is paired with. These previous environments may be needed when a query increment fails and an earlier state of the accumulated query has to be reconstructed.

The remaining explanations appear as comments in the following listing of a Prolog program for answering incremental queries.

```
iq(Env,Stack) :-
        read_query(Input,Query,Env),
                % A query increment is read. Input is the form in
                % which it is entered by the user. Query is the
                % result of translating its variable names (regarded
                % as constants) to system-supplied variables, using
                % Env. When read_query succeeds, Env is usually a
                % further instantiation compared to what it was when
                % read_query was called. The difference is in new
                % variables that may be present in Input.
        (Query=ok,!
        ;
         (solve_query(Query,Input,Env,Stack,New_Env,New_Stack),
                % Solves the accumulated query up to Query,
                % adding the answer substitution to Env, giving
                % New_Env, and adding Input and a back-up copy
                % of Env to Stack, giving New_Stack.
          iq(New_Env,New_Stack)
         )).
solve_query(Query,Input,Env,Stack,New_Env,New_Stack) :-
        save_environment(Input,Env,Input+Env1),
        solve(Query,Env,Stack,Input+Env1,New_Env,New_Stack),
                % Answers Query, using existing bindings in Env
                % and using Stack as back-up. Stack is the back-up
                % up to but not including Query. Input+Env1 is
                % added to Stack, giving New_Stack as new back-up.
                % New_Env contains the old bindings of Env plus
                % any new ones created by Query.
        display_solution(New_Env).
solve(Query,Env,Stack,Input+Env1,Env,[Input+Env1|Stack]) :-
        call(Query).
                % Query can be solved in the current ENV.
                % Note that when call succeeds, Env is usually
                % a further instantiated version compared to what
                % it was when solve was called. The difference is
                % in the bindings created by the success of call(Query).
```

```
solve(Query,Env,Stack,Input+_,New_Env,
                        [Input+Env1,Input2+Env2|New_Stack]) :-
                % Query can not be solved using current Env.
        backtrack(Input,Stack,New_Env,[Input2+Env2|New_Stack]),
                % First it backtracks to the previous query, the
                % answer of which is the cause of current failure.
                % Then it re-answers the queries as it
                % reconstructs the back-ups again. New_Env is a new
                % answer substitution.
        extract_partial_ans(New_Env,Env2,Env1,Input).
                % Extracts the back-up environment for the query
                % Input from New_Env.
backtrack(Input,[Input1+Env1|Stack],Env1,[Input1+BackEnv|Stack]) :-
                % Pops up a previous query Input1 and its back-up
                % environment Env1 from Stack. Note that
                % generally Input is a conjunctive goals consisting
                % of the last query and the previous queries already
                % popped up from the Stack.
        construct_queries(Input,Input1,Env1,Queries,BackEnv),
                % Constructs conjunctive goals, Queries, from Input
                % and Input1 based on the back-up environment Env1
                % of Input1.
        call(Queries).
                % Solves the conjunctive goals. If it succeeds, Env1
                % is a new answer substitution for the whole query.
backtrack(Input,[Input1+_|Stack],AnsEnv,
                        [Input1+Env1,Input2+Env2|New_Stack]) :-
                % Input and Input1 can not be solved simultaneously.
        app_token(Input,Input1,Inputs),
                % Constructs Inputs by adding Input1 to Input.
        backtrack(Inputs,Stack,AnsEnv,[Input2+Env2|New_Stack]),
                % backtrack is recursively called with Inputs as its
                % new first argument.
        extract_partial_ans(AnsEnv,Env2,Env1,Input1).
                % Now AnsEnv is a new answer substitution for the
                % whole query. What remains to be done is to reconstruct
                % back-up environment for each queries popped up from
                % Stack. Here the back-up environment Env1 of
                % Input1 is extracted from AnsEnv.
```

In the appendix, we show a version where a form of intelligent backtracking is implemented.

Incremental queries were originally conceived as solution to the problem of doing computer graphics interactively, yet without appeal to side effects. More generally, incremental queries allow one to interact with computations where the user enters transactions interactively with the results of previous transactions. Examples are data-base transactions or interactions with a processor for a language like Basic. Sergot [3] has suggested that

incremental queries are also useful for another kind of interaction: where the user requires as result of a computation an object satisfying many constraints and where he does not know explicitly and in advance what the constraints are.

Existing Prolog implementations effectively require the user to renounce the right to enter more constraints in exchange for being allowed to see a solution. With incremental queries, however, the user can ask at any stage for a solution to the query as accumulated so far, while retaining the option to add further goals (i.e. constraints). When he sees the first solution to the query accumulated so far he may see a feature he does not like. This will prompt him to enter as next increment a goal eliminating the unacceptable feature. The above program will then be forced to backtrack and present, if available, another, improved, solution. This accommodates a very important and common property of users: *they do not know in advance what they want, yet afterwards they do know what they don't want.*

The following example demonstrates the aspect of incremental queries just discussed. The problem is to arrange a committee meeting in such way to satisfy constraints, for example, on members who must attend, and on the total number of the members attending. We use an assertion of the form

```
available (Member, Day)
```

to state that Member is available on Day. Suppose the committee has eight members: a,b,c,d,e,f,g, and h.

```
available(a,1).  available(a,3).  available(a,5).
available(b,2).  available(b,3).  available(b,4).  available(b,5).
available(c,1).  available(c,4).
available(d,2).  available(d,4).  available(d,5).
available(e,2).  available(e,5).
available(f,2).  available(f,4).
available(g,1).  available(g,2).  available(g,3).  available(g,5).
available(h,1).  available(h,2).  available(h,3).  available(h,4).
available(h,5).
```

Let us arrange the committee meeting.

?- iq.

% Member a has to attend the meeting.

??- available(a,Day).

Day = 1

% Member b has to attend the meeting, too.

???- available(b,Day).

Day = 3

% Who will be able to attend the meeting on Day ?

???- setof(Member,available(Member,Day),Members).

Day = 3
Member = _1491
Members = [a,b,g,h]

% How many members will be able to attend the meeting on *Day* ?

???- length(Members,N).

Day = 3
Member = _1491
Members = [a,b,g,h]
N = 4

% Four is not enough. At least five members have to attend.

???- N >= 5.

Day = 5
Member = _2746
Members = [a,b,d,e,g,h]
N = 6

% Six members is enough to open the meeting. OK.
% The meeting will be opened on the 5th.

???- ok.
??- end.

The example demonstrated that the user need not know all the constraints in advance: with incremental queries, the preliminary solutions will remind him when it becomes necessary for a constraint to be stated. It may happen that the user is blissfully unaware of the need for a constraint when he forgets to enter it and, because of a lucky ordering of solutions, the first solution he gets satisfying the constraints he thought of, also satisfies the ones he forgot about.

It may of course also happen that the constraints entered eliminate all solutions. Even then, not all may be lost: the user may be willing to drop an earlier constraint. It would be unfortunate if all remaining constraints would have to be entered again. That is why we also implemented a facility allowing cancelling of queries entered earlier. See section 5 for a description.

3.3 A spreadsheet interface based an incremental queries

Our spreadsheet interface for logic programs consists of an incremental query facility coupled with a matrix display. As in conventional spreadsheets, each cell in the matrix is identified by the names of its row and by the name of its column. According to our model, each cell corresponds to a variable of a query. Whenever such a variable is instantiated, the corresponding matrix cell displays the value.

Thus our approach differs from Kriwaczek's [5], the only other work we know of combining spreadsheets with logic programming. We use the matrix display to present an answer substitution, whereas Kriwaczek uses it to present the content of a database of assertions, one for each non-empty cell, stating what the contents of that cell are.

Rather than list all of our program, we give a brief overview of this experimental implementation. The code shown so far consists of a few dozen lines. The actual program has about 250. The greatest part of the code is taken up by lower level utilities already necessary for running even the rudimentary portion listed in this paper. Another reason for the bulk of the actual program is that it has features not yet discussed.

To implement a matrix display we needed about another approximately 500 lines of Prolog code. A DEC VT131 connected to a timesharing computer running Edinburgh Prolog is not ideal for programming screens. This explains to a large extent the difficulty of this part of our implementation. However, we are glad we did it because it gave us valuable insights about the power of Prolog in combination with a spreadsheet interface.

When solving interactively a cryptarithmetic puzzle and the four-queens problem, the matrix display in combination with the incremental query gave us a much clearer picture of the state of the problem and what to do next than any Prolog interface we know. The ability to enter not just spreadsheet commands such as assignments, but any Prolog goal was essential here. For example, in both puzzles we need constraints saying that a new entry must be different from those found so far. This was easily done by entering the goal requiring that the entry not be a member of a list of such entries.

We have shown the basic part of the spreadsheet interface based on incremental queries. In the following sections we will add new features to the spreadsheet. One is automatic resolution of a certain simple type of conflict among queries. Another is cancelling of previous queries. In section 4, by restricting a query type to an equation, we will show that TYPICALC generalized in section 2 can be derived as a subset of our spreadsheet based on incremental queries. In section 5, we will show that the spreadsheet with the facility of cancelling previous queries is a useful aid to man-machine interaction in Prolog-based problem solving based on Prolog.

4 TYPICALC as a subset of Prolog augmented by a spreadsheet interface

So far, the only departure from the conventional queries of Prolog has been to give the user feed-back in the form of solutions to incomplete queries so that query continuations can be determined by the information thus obtained. What has not changed is that failure remains failure: if a conventional query fails, then so will its incremental version.

We now consider the added feature allowing the user to react to failure by reconsidering goals entered at an earlier stage of the interaction. Such a goal is replaced by another one and the remaining part of the query is solved again. Our main reason for studying this feature is its utility in combination with the requirement that all goals are equations of a restricted format.

Each equation we require to have an unbound variable on the left-hand side. The right-hand side has to be a term instantiated to be a ground instance at the time the equation is entered. We call these the TYPICALC restrictions for reasons that will become clear presently. Suppose that a query

$$e_1, \ldots, e_n$$

satisfying these restrictions is solvable and that

$$e_1, \ldots, e_n, e_{n+1}$$

is not. It must be the case that among $e_1, \ldots, e_n$ there is an equation e_i having the same left-hand side as e_{n+1}. We have implemented a processor for incremental queries that replaces in these circumstances e_i by e_{n+1} and then solves the modified accumulated query thus obtained.

The spreadsheet interface is especially attractive for such a query processor. When every variable corresponds to a cell on the screen, there is a handy way of representing an equation satisfying the TYPICALC restrictions: write the right-hand side in the cell corresponding to the left-hand side. The TYPICALC restrictions ensure that each cell gets only one entry. The only way a query can become unsolvable is by equating a variable to a value different from the value equated with the same variable in an earlier increment. Our processor interprets this as the user having changed his mind about the value of that variable and modifies the query as explained above. The modified query is then evaluated again. As a result, all variables (hence cells) depending on the changed variable, change their values accordingly. Indeed our prototype spreadsheet interface shows the famous TYPICALC effect: giving a cell a new value causes all dependent cells to change accordingly.

Let us now consider an elaboration of the last program in section 3 (let us call it Version 3) that will allow queries to be modified in this way. In Version 3, `solve_query` fails if `solve` does. The additional facility of allowing queries to be modified should therefore be accommodated as another clause for `solve_query` to be placed after the existing one.

```
solve_query(_,Input,Env,Stack,New_Env,New_Stack) :-
        equation(Input,Variable),
        % Input has been recognized as an equation with Variable
        % as left-hand side.
        save_environment(Input,Env,Input+Env1),
        resolve_conflict(Variable,Input+Env1,Stack,New_Env,New_Stack),
        display_solution(New_Env).
resolve_conflict(Variable,Input+Env,Stack,New_Env,New_Stack) :-
        back_to_conflict(Variable,Input+Env,Stack,Before,[],
                         [Input1+Env1|After]),
        re_solve_after(Before,[Input1+Env1|After],Env1,New_Env,
                       New_Stack).
```

```
back_to_conflict(Variable,Input0+_,[Input+Env|Before],Before,After,
                 [Input0+Env|After]) :-
        equation(Input,Variable),!,
        % Input is recognized as equation and Variable is matched to
        % its left-hand side. Hence it is replaced by the equation
        % specified in Input0.
        report_removed_query(Input).
back_to_conflict(Variable,Input0+Env0,[Input+Env|Stack],Before,Work,
                 After) :-
        % When this clause is used, Input is not an equation or not
        % with Variable as left-hand side.
        back_to_conflict(Variable,Input0+Env0,Stack,Before,
                         [Input+Env|Work],After).
re_solve_after(Stack,[Input+_|Unsolved],Env,New_Env,New_Stack) :-
        save_environment(Input,Env,Input1+Env1),
        construct_query(Input+Env,Query),
        solve(Query,Env,Stack,Input1+Env1,NEnv,NStack),
        re_solve_after(NStack,Unsolved,NEnv,New_Env,New_Stack).
re_solve_after(NStack,[],NEnv,NEnv,NStack).
```

5 Constraint cancellation: an additional tool in interactive problem-solving

To summarize, our spreadsheet interface augments conventional Prolog in the following two aspects.

(1) *It is more interactive:*
Of course, conventional Prolog should definitely be considered an interactive language when classified among all other programming languages. Yet, it can be made considerably more interactive: queries are, as it were, prepared and processed *in batch* - only when a query is complete, is a user allowed to see a solution. Our incremental queries make Prolog *more interactive* by allowing preparation of queries interactively with preliminary solutions.

(2) *It is more visually oriented*
The two-dimensional arrangement of variables seems to be natural to many applications. These include, as is widely appreciated, the clerical applications representative of TYPICALC. We suspect the same holds for many other applications, as exemplified by standard toy problems such as cryptarithmetic and n non-attacking queens.

In this section we argue for an additional tool in interactive problem-solving: namely the facility of *cancelling goals* previously entered in an incremental query. Up till now we have only focussed on one imperfection of users: the inability to be aware in advance of all relevant constraints. Let us now consider the opposite weakness, of wanting too much: the constraints entered (which may be premeditated or prompted by unacceptable preliminary solutions) may rule out *all* solutions. The user may then want to enter into *bargaining mode*: to give up an earlier constraint and see what solutions he gets in return.

We implemented a variant of the incremental query facility described above to allow

constraint cancellation. It differs from the original version as follows

(1) When the goals entered so far become unsolvable, the system prints out the whole query.
(2) The user can remove a previous goal by *remove(<number>)* command, where *<number>* is a number associated with each goal by the system.

To demonstrate interactive problem solving using goal cancellation, the previous example is re-visited.

```
?- iq.
        % Member a has to attend the meeting.
??- available(a,Day).
Day = 1
        % Member c has to attend the meeting, too.
???- available(c,Day).
Day = 1
        % Who will be able to attend the meeting on Day ?
???- setof(Member,available(Member,Day),Members).
Day = 1
Member = _1140
Members = [a,c,g,h]
        % How many members will be able to attend the meeting on Day ?
???- length(Members,N).
Day = 1
Member = _1140
Members = [a,c,g,h]
N = 4
        % Four is not enough. At least five members should attend.
???- N >= 5.
I can not solve the following queries.
                        % The system automatically prints out
                        % all the queries.
------------
[0] N >= 5
------------
[1] length(Members,N)
------------
[2] setof(Member,available(Member,Day),Members)
------------
[3] available(c,Day)
------------
[4] available(a,Day)
```

```
        % Member c is essential, but a perhaps not.
        % So we remove the goal specifying that a has to attend.
???- remove(4).
        % The system removes the specified goal and re-solves
        % the remaining goals.
Day = 4
Member = _3599
Members = [b,c,d,f,h]
N = 5
        % Good enough!
???- ok.
??- end.
```

In the above example, a user cancelled a previous goal by a *remove* command, and he got a satisfactory result. It is worth noting that the variables such as *Day* and *Members* are used like global variables to communicate between query increments.

6 Conclusion

What constitutes an advance in user interfaces? We feel it lies in the choice of a suitable *metaphor* to organize information [4]. For example, a screen-oriented editor uses a two-dimensional sheet of paper as metaphor familiar to the user. As such it is an improvement over line-oriented editors (Where the metaphor is perhaps the slit through which a tank commander views the world). Another example is the use of a desk top as metaphor, including icons for file folders, documents, a trash can, etc., perhaps most widely known as user interface of the Apple Macintosh computer. And, of course, spreadsheets owe their success to being a metaphor, improving over files that can only be accessed in batch processing by update programs or report generators.

The strength of logic programming lies in its contributing a new, powerful metaphor: it gives a user the illusion of a conversational partner with whom questions, answers, rules, facts, and explanations can be exchanged. However, all this is situated above the level of the mechanics of the interaction. At this lower level, conventional Prolog implementations are as crude as line-oriented editors. At this level an independent metaphor is required. We believe to have found a suitable one by borrowing matrix displays from spreadsheet programs and supporting them by incremental queries.

Independently of their essential role in our use of the spreadsheet metaphor, incremental queries play an important role in interactive problem-solving. They encourage the user to think in terms of successive approximation to a solution by a sequence of constraints built up *in parallel* with the sequence of approximations. This parallelism is essential, as there may be feed-back from the approximations to the constraints. In conventional logic programming, the role of a query is typically similar to that of a procedure call in other interactive languages: just to activate (with suitable parameters) a computation specified in advance. With incremental queries, we expect the Marathon Query to become commonplace: an ad-hoc one that goes on and on, taking unexpected turns, creative in itself rather than

a mere activation of a predefined computation.

Acknowledgments

We wish express our thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity of this research in the Fifth Generation Computer Systems Project at ICOT. We would also like to thank Hiroyasu Kondou and other ICOT research staffs.

References

[1] M.H. van Emden: Logic As An Interaction Language. Proc. 5th Conf. Canadian Soc. for Computational Studies in Intelligence (1984), pp126-128.

[2] D.L.Bowen, L.M.Pereira, F.C.N.Pereira and D.H.D.Warren: User's guide to DECsystem-10 Prolog. Dept of Artificial Intelligence, University of Edinburgh (1282).

[3] M. Sergot: Private communication.

[4] P. Heckel: The Elements of Friendly Software Design. Warner Books, 1984.

[5] F. Kriwaczek: LogicCalc - A Prolog Spreadsheet. to appear in Machine Intelligence 11, D. Michie and J. Hayes (eds.).

Appendix: Intelligent backtracking of incremental query

When an incremental query is applied to a spreadsheet, intelligent backtracking becomes more important, since in spreadsheet applications many goals may be independent and it is necessary to avoid redundant computation to make the system practical.

The form of intelligent backtracking we implement only considers the dependency relation between the goals entered by a user. Therefore it is simpler than general intelligent backtracking. If the underlying Prolog system has intelligent backtracking, our restricted intelligent backtracking is not necessary. However, the restricted intelligent backtracking is adequate for incremental queries on top of ordinary Prolog system.

In our system, the dependency relation between goals is detected when the goals share more than one variable, since when one of the queries is re-solved, then the rest of the goals may change their answers. The dependency relation considered here is defined as follows:

Definition. The dependency relation

When two goals, say Q1 and Q2, satisfy at least one of the following two conditions, then the two goals are regarded as being in the dependency relation.

1. The two goals, Q1 and Q2, share more than one variable.
2. There exists a goal, say Q, with which both Q1 and Q2 have the dependency relation.

Note that the dependency relation defined above is an equivalence relation. Therefore we can classify queries into several equivalence classes with respect to the dependency relation, so that queries in the same class are mutually dependent and any two goals in the different classes are independent.

A program for incremental queries is now augmented to keep one more item of information, that is, the dependency list, in order to realize the intelligent backtracking. An element of the dependency list is a pair of a list of variable names and a list of query numbers called a dependency stack. The latter corresponds to the equivalence class mentioned above and is ordered chronologically, and the former includes the variables which appear in the goals in the latter. The management of the dependency list is illustrated in the following example.

Query	**Stack***	**Dependency List**
(1) int(A1).	[int(A1)]	[(['A1'],[1])]
(2) int(A2).	[int(A1),int(A2)]	[(['A1'],[1]),(['A2'],[2])]
(3) int(A3).	[int(A1),int(A2),int(A3)]	[(['A1'],[1]),(['A2'],[2]), (['A3'],[3])]
(4) A1=A2.	[int(A1),int(A2),int(A3), A1=A2]	[(['A1','A2'],[1,2]), (['A3'],[3])]
(5) A2=A3.	[int(A1),int(A2),int(A3), A1=A2,A2=A3]	[(['A1','A2','A3'],[1,2,3])]

*(**Stack** is similar to Stack in the previous program, but simplified.)

The definition of the solve predicate shown in the previous section is changed to the following. The solve predicate first updates the current dependency list, Dep_list, to New_Dep_list by examining the dependency relation between the current goal and the previous goals, and calls the solve1 predicate which corresponds to the solve predicate in the previous program.

```
solve(Query,Env,Stack,Dep_list,Num+Input+BackupEnv,
                        New_Env,New_Stack,New_Dep_list) :-
                % Num is the query number.
        get_variable(Input,Var_list),
                % Extracts a list of variable names, Var_list, from
                % the input query, Input.
        make_dep(Num,Dep_list,Var_list,New_Dep_list,Dep_stack),
                % Updates the dependency list.
                % Dep_stack is a dependency stack including
                % the current query.
        solve1(Query,Env,Dep_stack,Stack,Num+Input+BackupEnv,
                        New_Env,New_Stack).
                % Solves the query, Query. Dep_stack is
                % the dependency stack to which Query belongs.
```

The definition of the solve1 predicate is similar to that of the solve predicate explained in the previous section. They differ in two respects. One is that the solve1 predicate has one additional argument for the dependency list. The other is that, in Stack, each element is associated with a number, which identifies a query.

```
solve1(Query,Env,_,Stack,Num+Input+BackupEnv,Env,
                        [Num+Input+BackupEnv|Stack]) :-
        call(Query).
solve1(_,Env,Dep_stack,Stack,Num+Input+_,New_Env,
                        [Num+Input+Env|New_Stack]) :-
        backtrack(Dep_stack,Stack,Input,New_Env,New_Stack,
                        Input1,Env1,Env),
                % The current environment, Env, and the dependency
                % stack, Dep_stack, are passed to the backtrack predicate.
        extract_partial_ans(New_Env,Input1,Env1,Input,Env).
```

The backtrack predicate is also changed to have two additional arguments, Dep_stack and Env. Dep_stack is used to pass the dependency stack, since now backtracking occurs only in the dependency stack. Env is used to pass the current environment, which includes the answer of the queries which are not in Dep_stack.

```
backtrack([Num|Dep_Stack],[Num+Input1+Env1|Stack],Input,Env1,
                [Num+Input1+BackEnv1|Stack],Input1,BackEnv1,_) :-
                % If the goal number on top of the dependency stack
                % is equal to that of the goal, Input1, on top of the stack,
        construct_queries(Input,Input1,Env1,Queries,BackEnv1),
                % Makes the conjunction of Input1 and Input,
        call(Queries).
                % Tries to solve it.
```

```
backtrack([Num|Dep_Stack],[Num+Input1+_|Stack],Input,AnsEnv,
                [Num+Input1+Env1|New_Stack],Input1,Env1,Cur_Env) :-
                % If the query number on top of the dependency stack
                % is equal to that of the query, Input1, on top of the stack,
                % and the conjunction of Input1 and Input is unsolvable,
        app_token(Input,Input1,Inputs),
                % Makes the new Inputs by literally appending
                % Input and Input1.
        backtrack(Dep_Stack,Stack,Inputs,AnsEnv,
                                New_Stack,Input2,Env2,Cur_Env),
                % Further backtracks through the dependency stack with
                % Inputs as the new input.
        extract_partial_ans(AnsEnv,Input2,Env2,Input1,Env1).
                % Generates the new back-up Env1.
backtrack([Num0|Dep_Stack],[Num+Input1+_|Stack],Input,AnsEnv,
                [Num+Input1+Env1|New_Stack],Input1,BackEnv1,Cur_Env) :-
        Num0 = Num, !,
                % If the query number on top of the dependency stack
                % is not equal to that of the query, Input1, on top
                % of the stack, it means that Input1 is not in the
                % dependency stack and need not to be solved again.
        backtrack([Num0|Dep_Stack],Stack,Input,AnsEnv,
                                New_Stack,Input2,Env2),
                % Just skips Input1 and continues backtracking.
        extract_partial_ans(AnsEnv,Input2,Env2,Input1,Env1),
                % Generates new back-up, Env1, for Input1.
        restore_independent_ans(Cur_Env,AnsEnv,Input1).
                % Restores the answer of Input1 from Cur_Env to AnsEnv.
```

A comparison between the efficiencies of the new `solve` predicate and the ordinary Prolog interpreter is shown below, where `int(N)` is a predicate which succeeds if and only if N is a digit. The definition of `int` is:

```
int(1).  int(2).  int(3).  int(4).  int(5).
int(6).  int(7).  int(8).  int(9).  int(0).
```

Goal statement:

```
int(A1),int(A2),int(A3),int(A4),int(A5),A1=2.
```

(1) Incremental query with intelligent backtracking
(Goals are entered incrementally one by one.)
CPU time = 123 ms.

(2) Prolog interpreter
CPU time = 4213 ms.

In the goal statement, goals are classified into five classes, that is, {int(A1),A1=2}, {int(A2)}, {int(A3)}, {int(A4)}, {int(A5)}. In case of the incremental query with intelligent backtracking, when the last increment, A1=2, is entered, the system only re-solves

the goal, `int(A1)`. However, Prolog interpreter backtracks from `int(A5)` to `int(A1)`. It demonstrates the efficiency of the proposed intelligent backtracking introduced to a program of incremental queries.