Direct manipulation of Algebraic Formulae in Interactive Proof Systems

Yves Bertot

INRIA Sophia Antipolis

Abstract. In this paper we present a tool that uses the mouse to quicken some frequent operations on mathematical formulae. This tool uses a paradigm of object dragging to replace expressions by provably equivalent formulae, using a *rewrite* command. The paper also shows how the tool can be programmed to take new operators into account.

1 Introduction

Rewriting is a reasoning technique that is especially easy and lively to perform at a blackboard, where large expressions can be manipulated bit by bit rubbing out the transformed expressions and replacing these with equivalent expressions (sometimes with an oral justification for the simplification, indicating a number is non zero or strictly positive). Rewriting has been used extensively in computerized proof systems and computer algebra systems for a long time. However, the traditional mode of interaction with computers, based on requests and answers, has always been an obstacle to the kind of direct manipulation that is customary in "blackboard" reasoning, and thus automatic rewriting has always prevailed. Still, the advent of graphical interfaces should make it possible to use the computer screen as an "intelligent" blackboard, performing the rewriting requested by the user and checking (or recording for later treatment) the justifications needed for complex simplifications.
An original idea is to view some operations as movement of data on the screen. This idea has been first implemented in a computer algebra system like Theorist [7] (distributed as MathPlus in Europe), where the use of the mouse to move around sub-formulae in mathematical expressions has been integrated in a graphical user-interface with some success. The tool takes advantage of specific properties of known algebraic operators, like addition and multiplication, to provide easy ways to permute terms in an addition, to eliminate equal terms on both side of an equality, and to help solve equations by isolating the desired unknown. The work described in this paper continues this trend in the realm of proof systems where justifications have to be provided in the form of a proven equality.
Section 2 of this paper describes how the drag-and-drop facility integrates with an interactive proving environment. Section 3 describes the basic engine that makes this tool possible. Section 4 describes how the user can extend the tool to adapt it to the mathematical domain of their choice. Section 5 concludes this paper with a rundown on the current implementation and ideas for further extensions.

2 The end-user view of drag-and-drop rewrite

Drag-and-drop rewrite proceeds by performing little transformations in large mathematical formulae. The main achievement of drag-and-drop rewrite comes in the way users can specify the operation to perform, with an intuitive comprehension of the operations involved in the process.
The user-interface that hosts the drag-and-drop tool provides a graphical window where mathematical formulae are displayed. Operations triggered by the user provoke this window's contents to change

in: *User Interfaces for Theorem Provers (UITP)*, Sophia Antipolis, France, September, 1997.

accordingly. The departure from traditional interaction facilities is that the graphical layout of the mathematical formula itself is used to communicate the next operation from the user to the symbolic engine. To perform an operation, the user simply points at an expression and indicates where this mathematical expression has to go. In this sense, the drag-and-drop rewrite tool makes it possible to *tell* the computer how to proceed the same way as a math teacher would tell a student by simply pointing at a blackboard. Thus, the whole point of our experiment is to find a way to interpret movements provided by the human user.

2.1 Example

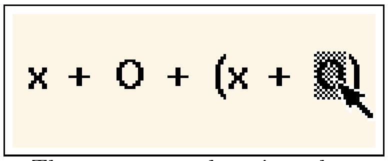
Let us suppose the current goal to prove contains a logical formula of the form:

$$(x + 0) + (x + 0).$$

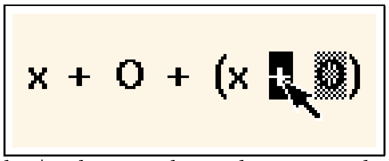
The user can select the second (rightmost) 0 and drag the mouse to the closest + sign (the third one). The proof system generates and executes a command that makes a new goal appear, where the previous expression is replaced with the following one:

$$(x + 0) + x$$

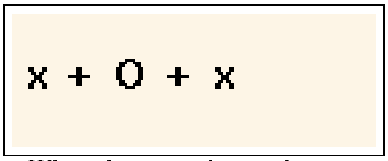
Note that the first instance of $x + 0$ is not modified. Intuitively, the gesture has been interpreted as a request to use the properties of 0 with respect to addition only at the designated place.



a. The user starts dragging, the starting position changes color.



b. As the user drags the mouse, the final position is displayed with a different color.



c. When the user releases the mouse, a rewrite operation is performed.

Fig. 1. An example of drag-and-drop behavior

2.2 Data capture

Movements are only interpreted in terms of the *start* and *finish* positions. The user can indicate this positions with the mouse.

The graphical layout of mathematical formulae can be used for various functionalities, among which drag-and-drop rewrite is only one possibility. In general the dragging movement is used to select sub-formulae, not for the drag-and-drop mechanism. To distinguish uses of the drag-and-drop mechanism, our user-interface uses a key modifier, the `control` key. When the control key is depressed and the user selects an expression, this expression automatically changes to a specific color. As they move the mouse with both the control key and the mouse button still depressed, a second colored selection appears. The two colors indicate a *start* position and a *finish* position for the drag-and-drop rewrite tool.

2.3 A taxonomy of mathematical movement

In general, the drag movement is used to express one of several kinds of operations:

1. The combination of two expressions on the screen, such as the two x's in the expression $x - x$ (and replace it by 0, this will be expressed by taking one of the x's and dragging to the other one).
2. The movement of some expression with respect to other expressions, such as when one wants to replace $x + y$ by $y + x$ or $x + (y + z)$ by $y + (x + z)$. In both cases this will be expressed by taking the x and moving it to the y.
3. The cancelation of several operations together, $-(-x)$ which can be replaced by x. This will be expressed by taking one of the $-$ signs and moving it to the other.
4. The permutation of operators such as in $-(1/x)$ which can be replaced by $1/(-x)$. This will be expressed by taking the $-$ sign and moving it on the $1/$ sign.
5. The rearrangement of complex structures like associative patterns of the form $(x + y) + z$, which can be replaced by $x + (y + z)$. This will be expressed by taking the parentheses and and moving them to the second "plus" sign, to express that we want the parentheses around the second sum.

2.4 Comparison with the traditional mode of interaction

When no mouse interaction is provided, the user must take care of three aspects when precise rewrite operations are needed to push the proof forward. First, the name of the relevant rewrite theorem must be found. Second, it may be necessary to give the precise instantiation of the theorem that will be used because several occurrences of the patterns involved in the rewrite theorem may occur in the goal. Last, it may also be necessary to limit the scope of the rewriting so that only the chosen instance in the goal is modified during the operation. Among these aspects, the need to limit the scope may be the most cumbersome: when mathematical formulae become large, it is difficult to provide unambiguous ways to refer to a precise instance of a command.

Drag-and-drop rewrite provides automatic support for these three aspects. First, the location where the user pointed at is directly used to determine the rewrite theorem to apply. Second, the precise instantiation of the theorem is directly computed from the data on the screen. Last, the scope is automatically limited to the location the user pointed at. Thus, a rewrite command can be requested in one gesture instead of a long search for information.

On the other hand, the quick interaction also has its limitations. The drag-and-drop tool only applies to certain classes of rewrites, those where the operation can intuitively interpreted as a movement of data or as the marriage between expressions. Also, when several theorems apply to one given pattern, it will always be the same theorem that will be applied. In this sense, drag-and-drop rewrite must only be used as a complement to other tools that enable rewriting. For instance, the cHOL system [5] provides a tool that selects all the rewrite theorems that apply to a given pattern. In the user-interface we use as a test-bed [1], mouse interaction is already provided in the form of *proof-by-pointing* [2] and *point-and-shoot* extensions [3]. Other paradigms that use the mouse to limit the scope of operations include *window inference* [4].

3 Implementation and mode of operation

3.1 Basic data structures

Our implementation uses a direct representation of mathematical formulae as trees, with a layout mechanism that displays these trees on the screen and automatically computes the selected tree when one gives a position on the screen. This selected tree is referred to via a navigation path in the tree structure.

Navigation paths can be viewed as lists of integers. In the following we will denote the concatenation of two paths as $p \cdot q$. Manipulations on these paths are coherent with manipulations on trees and contexts. If p is a valid path in the tree T, then we can denote by T/p the subterm designated by the path p in T and by $T[p \leftarrow t]$ the tree obtained by replacing this subterm by the tree t. An important coherence property is that subterm and path concatenation commute:

$$T/(p \cdot q) = (T/p)/q$$

and similarly subterm replacement and path concatenation commute:

$$T[(p \cdot q) \leftarrow t] = T[p \leftarrow ((T/p)[q \leftarrow t])].$$

The tree structures we consider may contain special trees called *metavariable* (the prefix *meta* is used to distinguish these variables from the mathematical variables used in mathematical formulae to represent unknowns or parameters). These metavariables come in conjunction with a pair of primitives `match` and `substitute`. The `match` primitive takes two trees and returns the list of metavariable bindings when the second tree is an instance of the first one, otherwise it fails. The `substitute` primitive applies a list of metavariable bindings to a tree. The two primitives are bound by the following equation, valid as soon as $\texttt{match}(t, t')$ is defined:

$$\texttt{substitute}(t, \texttt{match}(t, t')) = t'$$

The `match` primitive provides nonlinear matching, in the sense that several occurrences of the same metavariable must correspond to the same value in the matched tree. In other words, if a metavariable occurs in a tree t at two positions denoted by the paths p_1 and p_2, $\texttt{match}(t, t')$ will succeed only if $t'/p_1 = t'/p_2$. In the following, trees containing metavariables will often be called *patterns* following the tradition in the study of first-order rewriting and first-order unification.

3.2 Drag-and-drop rules

Drag-and-drop rules are tuples of the form $(\mathcal{F}, p_f, p_l, \mathcal{C}, n)$, where $\mathcal{F}$ is formula pattern, p_f is a valid path in $\mathcal{F}$, used to represent the first position in a drag-and-drop gesture, p_l is a valid path in $\mathcal{F}$, used to represent the last position in the gesture, $\mathcal{C}$ is a command pattern[1], and n is a rule name, used to make it easier to refer to the rule. The command pattern $\mathcal{C}$ is necessarily a rewrite command of the form

$$\texttt{Rewrite (some\_theorem } meta_1 \texttt{ ... } meta_n\texttt{)}$$

or

$$\texttt{Rewrite <- (some\_theorem } meta_1 \texttt{ ... } meta_n\texttt{).}$$

In this pattern, the theorem `some_theorem` should have a conclusion of the form $\mathcal{A} = \mathcal{B}$ where $\mathcal{F}$ is an instance of $\mathcal{A}$ (or $\mathcal{B}$ if the command pattern has the second form). Note that the theorem may express a conditional equality, in the sense that it may require some hypotheses to be fulfilled. The proof of these hypotheses will simply be requested as extra goals.

[1] In the user-interface we used as a test-bed, tree structures are not only used for mathematical formulae, but also for the commands sent to the proof engine.

3.3 The drag-and-drop engine

The drag-and-drop engine roughly operates in the following manner. The input data consists of the tree structure T corresponding to the mathematical formula displayed on the screen, two paths p_1 and p_2 describing the gesture of the user, and an ordered list of drag-and-drop rules. From this data, the engine computes the first matching drag-and-drop rule, instantiates its command pattern and sends it to the proof engine for execution. We now give more details on the way this drag-and-drop rule is determined.

First, the two paths p_1 and p_2 are used to compute a third path p_3, the longest common prefix to p_1 and p_2. Then for each rule $(\mathcal{F}, p_f, p_l, \mathcal{C}, n)$, we perform the following operations (please note that steps 2, 3, and 5 may fail).

1. Compute the longest common prefix p_r to p_f and p_l,
2. Compute the path p such that $p_3 = p \cdot p_r$,
3. check that $p_1 = p \cdot p_f \cdot p_1'$ and $p_2 = p \cdot p_l \cdot p_2'$ for some paths p_1' and p_2',
4. Let t be the subterm at path p in the mathematical formula T $(t = T/p)$,
5. Check that t is an instance of $\mathcal{F}$ and compute the list of bindings $\sigma = \mathtt{match}(\mathcal{F}, t)$,
6. Apply σ on $\mathcal{C}$, using the `substitute` primitive.

When some step fails, the engine simply goes on to the next rule. Also, step 1 can be performed once and for all when preparing the drag-and-drop rule, instead of performing it at each invocation of the drag-and-drop engine.

3.4 Interaction with the pretty-printing mechanism

We have already mentioned that tree structures are displayed on screen with a layout mechanism that automatically computes selected trees from positions on the screen. This computation uses the property that every displayed token has a reference to a sub-expression of the displayed tree. However, it is not the case that every sub-expression of the mathematical formula can be reached using the mouse. For instance, the pretty printer may be configured in such a way that the product of x and y is simply represented by the mere juxtaposition of the two expressions: xy. In this case, it is not possible to select the whole product by pointing to any particular token.

The drag-and-drop mechanism is designed to cope with this issue. In step 3, the tool does not check that $p_1 = p \cdot p_f$, but only that $p \cdot p_f$ is a prefix of p_1. Thus, if the position p_1 cannot be reached through a token on the screen, it will be enough to select a subterm of the desired term and this will produce a path that is longer than p_1.

An undesired side-effect of this policy is that some rules may become impossible to trigger: when p_f is a prefix of p_l and the pretty-printing mechanism prevents the user to select p_1 accurately, our drag-and-drop engine will compute the path p incorrectly. In practice, rules where one needs the two paths to be aligned are rare and we have decided to accept this weakness.

4 Extensibility

In this section we describe two tools provided to help users add their own drag-and-drop rules. The first tool is completely interactive and shows that the drag-and-drop functionality can be "programmed" by end-users in a way akin to "programming by example". The second tool is more automatic and is based on an analysis of equality theorems to detect their potential as drag-and-drop rewrite rules.

4.1 A rule editor

The simplest way to add a new drag-and-drop rule is to provide manually the formula pattern, the two paths p_f and p_l, and the command pattern. In our experiment, we provide a tool that makes this easy. This tool contains three windows: the first window contains the formula pattern, the second window contains the command pattern, the third window contains rule names. The first and second windows are equipped so to make it easier to replace a given subterm by a meta-variable.
To describe the paths p_f and p_l, the user simply needs to simulate the adequate movement in the window that contains the formula pattern. In this sense, the user simply gives an example of the movement that must captured by the drag-and-drop rule.
The order of drag-and-drop rules matters. After having edited a new drag-and-drop rule, the user should always consider the relative position this rule should have with respect to other rules. The third window is equipped with an automatic sorting algorithm that makes sure that general rules are place behind more specific ones. Still the user may need to adjust the respective positions of two rules whose patterns are not automatically comparable. Here again, this adjustment can be done by direct manipulation, using a drag-and-drop approach!

Example Let us suppose that the user just proved a theorem named `sample` with the following statement, where $(f\ x\ y)$ represents the application of the function f to x and y.

$$\forall x, y.(f\ x\ y) = (f\ y\ x)$$

The user wants this theorem to be applied if he drags the mouse from a to b in the expression $(f\ a\ b)$. To add this behavior, it is only necessary to perform the following steps:

1. Copy the expression $(f\ x\ y)$ in the test-bed interface's clipboard,
2. Open the rule editor and paste this expression in the first window,
3. Press the control key and drag the mouse from x to y in the first window,
4. Enter the command `Rewrite (sample x y)` in the second window,
5. Replace all occurrences of x with a metavariable `meta1` and all occurrences of y with a metavariable `meta2` in both windows,
6. Adjust the rule position in the rule list.

4.2 Automatic derivation of drag-and-drop rules

There are several classes of theorems for which one can introduce drag-and-drop rules with very systematic shapes. In our experiment, we propose automatic tools that will recognize when theorems belong to these classes and produce the corresponding drag-and-drop rules. The various classes recognized so far can be described as follows:

Cancelation theorems. Cancelation theorems have a conclusion of the form $x - x = m$, where the operator $-$ and the result m can vary. The drag-and-drop behavior associated to this kind of theorem is triggered when one selects one of the x's and drags to the other. The intuitive meaning is to combine the two instances of x.

Zero theorems. Zero theorems have a conclusion of the form $x + 0 = m$, $0 + x = m$, where the operator $+$ and the result m can vary (for instance the operator can be the multiplication operator and m can be 0). The drag-and-drop behavior associated to this kind of theorem is triggered when one selects the 0 and drags to the operator $+$. The intuitive meaning is to combine the 0 with the operation for which it has a special meaning.

Factorization theorems. Factorization theorems have a conclusion of the form $(x * y) + (z * y) = (x\%z)@y$, where the operators $\%$ and $@$ do not necessarily have to be the same as $*$ and $+$. Two drag-and-drop behaviors are associated to these kinds of theorems, corresponding to rewritings in both directions.

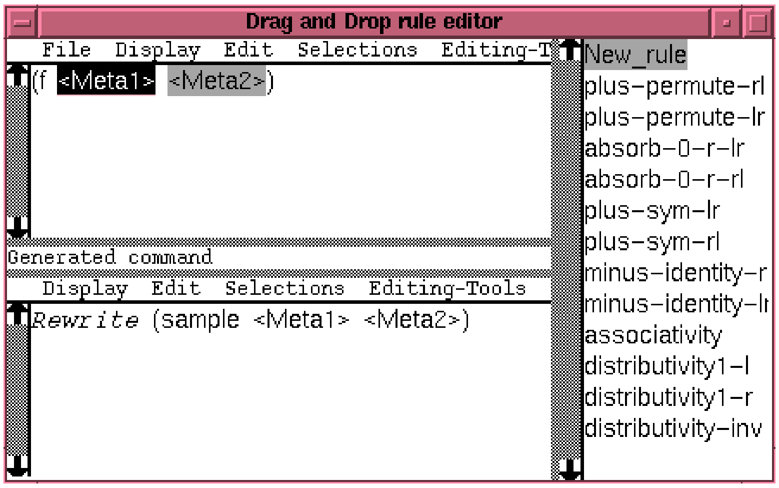


Fig. 2. The drag-and-drop rule editor. The top left window contains the matched pattern and indicates the start and end positions. The bottom left window contains the triggered rewrite command. The right window shows the ordered list of rules

1. If one finds an instance of $(x * y) + (z * y)$ and the user selects one of the y's and drags to the $+$ operator, this instance is replaced by the corresponding instance of $(x \% z) @ y$. The intuitive meaning is to drag the y's up the expression structure.
2. If one finds an instance of $(x \% z) @ y$ and the user selects the y and drags it to the x or the z, this instance is replaced by the corresponding instance of $(x*y)+(z*y)$. The intuitive meaning is to distribute the y to the arguments of the % operator.

Commutativity theorems Commutativity theorems have a conclusion of the form $x + y = y \% x$ where the operator % does not necessarily have to be the same as +, the drag-and-drop behavior associated to this kind of theorem is triggered when one selects x or y in an instance of the left-hand side or the right-hand side of the theorem and drags to the other argument.

In our test-bed interface, this procedure must be called explicitly by the user on the theorems of their choice.

5 Conclusion

This paper describes an interaction tool that makes it possible to perform frequent rewrite operations in a single movement using a *drag-and-drop* paradigm. This tool has been implemented in the CtCoq system [1] where it has been used to reason on polynomial expressions in an industrial application developed at Dassault-Aviation. The adaptation of this tool to a theory of real numbers required the capture of around one hundred drag-and-drop rules and this capture was performed as the theorems were proved. According to the engineer in charge of the industrial application, the drag-and-drop mechanism was a key factor in making the proof task tractable.

This drag-and-drop rewrite mechanism is an instance of the *equal opportunity* principle in user-interface design advocated in [6]. Direct manipulation implies that the output of the theorem prover is an important piece of data used in the generation of new input. Here, however, it is not so much the value

of the data manipulated that matters as its position. In this respect, this drag-and-drop facility is very similar to the proof-by-pointing and point-and-shoot facilities already studied by the author.
Unlike proof-by-pointing, this drag-and-drop mechanism exhibits no recursive behavior and only local modifications are performed for each gesture. For instance, in the presence of the formula $x + y + z + t$, it is not possible to replace it in one move with the formula $t + x + y + z$, although this would be a natural interpretation of the movement dragging t onto x. It seems that this limitation comes from the fact that our pattern matching mechanism does not provide a good way to use the associative-commutative properties of the addition. One may fear that a treatment that would give more importance to such properties might lessen the generality of the tool: how will it perform for operators that are not associative and commutative?
A second question pertains to the integration of drag-and-drop rewrite and proof-by-pointing. In our current implementation, the two tools are associated to two different mouse bindings (we use the third button of the mouse to perform proof-by-pointing and the control key with the first button to perform drag-and-drop). One may wonder whether there are simple ways to integrate the two tools in one interaction mode.
An other issue that is not covered in this paper is whether the tool is really natural and intuitive to use. One first remark is that the tool can easily be made very bad by triggering rewrite operations that have only little similarity with the gesture performed by the user. This is the main drawback associated to the fact that users can program their own behavior. Then, if we look at the various kinds of movements described in section 2.3, the movements that are the most natural seem to be the "real" movements, described in item number 2. The movements that feel less natural are those related to associativity described in item 5, because the modification in data can only with difficulty understood as a movement, especially if no parenthesis is displayed: the "intuitive" interpretation of the movement is that the data that moves is the pair of parentheses.
We have very little experience of how beginner users react to this tool. One of the drawbacks of our present setting is that a key modifier is needed to trigger the drag-and-drop facility. Beginner users have very little chance to use this capability spontaneously, without reading the documentation first. There are two ways to cope with this ignorance problem: a first way is to suggest the user to perform this operation as soon as one detects a situation where it could be applied (or randomly) a second way is to provide better integration with proof-by-pointing.

References

1. Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 231–234. Springer-Verlag, July 1996.
2. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.
3. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. To appear in Journal of Symbolic Computation.
4. Jim Grundy. Window inference inthe HOL system. In *Proceedings of the International Workshop on the HOL Theorem Proving System and its applications*, pages 177–189. IEEE Computer Society Press, August 1991.
5. Laurent Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic theorem proving and its applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.
6. Harold Thimbleby. *User Interface Design*. Frontier Series. ACM Press, 1990.
7. Waterloo Maple Sofware. *Math Plus, Learning guide and Reference manual*, 1995.