Implementing Proof by Pointing without a Structure Editor*

Yves Bertot†, Thomas Kleymann-Schreiber and Dilip Sequeira

Department of Computer Science
University of Edinburgh

October 27, 1997

Abstract: A proof by pointing user interface component allows a user to direct the course of a proof assistant by selecting terms with a mouse. Such a gesture is interpreted as a high-level tactical which triggers a sequence of low-level basic commands for the proof engine. The algorithm inherently relies on a structure-conscious environment; as a novelty we show how proof-by-pointing may easily be integrated into an interface without a structure editor. We discuss in detail the use of nested selectable text regions for user interaction, the modifications necessary to the proof-engine output, and the algorithm for interpreting selections as proof commands, with particular reference to a concrete implementation using XEmacs and LEGO.

1 Introduction

The issue of man-machine interaction for proof assistants has been recognised for a while. Many teams interested in proof system development have put significant efforts into the development of a specific man-machine interface. In some cases, communication through a graphical user-interface is the intended usage (see for instance the systems Nuprl [6], Alf [10], IPE [14], Jape [4]). In some other cases, a graphical user-interface is adjoined to a system which also provides a plain tty user-interface, based on parsing commands and printing out results (see for instance TkHOL [15] that uses the graphical toolkit Tcl/Tk [11] and CtCoq [1] and cHOL [16] that use a communication kit based on the programming environment generator Centaur [17]).

A sensible approach can also be to program a specific environment using a programmable text editor like Emacs [5]. Even though the programming environment can still be described as a *graphical* environment, since the editor can handle multiple fonts and colors, popup and pulldown menus, and mouse interaction, it is also fair to indicate that environments built this way are also adapted to work on plain terminals, as long as these

*also available as INRIA research report RR–3286

†INRIA Sophia-Antipolis

terminals are able to handle cursors. When considering Emacs-based environments, a wide range of elaboration can be achieved. Simple environments can provide help for keyword highlighting, command sending, and session recording, especially since the Emacs libraries provide a nice set of modules adapted for these tasks. Other environments can provide more specific features, with special windows for various kinds of data, menus that evolve with the state of the proof system, and so on. Systems like PVS [12] or IMPS [7] advertise especially their Emacs environment. An original case is Boomborg-PC [8], which is developed entirely in Emacs to present the task of system proving as the direct editing of proof terms in type theory.

One important issue in the development of these proof environments is the importance of structure editing versus text editing. Structure editing puts the emphasis on the tree structure of logical formulas and commands. It makes it possible to provide more elaborate notations and to give more help to the editor, in the form of menus which indicate the constructs that are allowed in the current context. On the other hand, structure editing puts constraints on the editing process, which some users may find unwieldy.

The paper [2] introduces a man-machine interaction technique called *proof-by-pointing*, which makes it possible to perform entire proofs in propositional logic using only the mouse to convey the intentions of the human user in a very efficient way. The intuitive idea of this technique is that the proof system displays the goals in a graphical window, and the user can guide the proof process by simply indicating the sub-expressions of these goals that are important. The proof system then executes commands that bring these sub-expressions to "the foreground" and applies trivial reasoning figures. The authors of [2] insist that the technique relies on a need for the editing component to be aware of the structure of logical formulas and they explain that their use of an environment based on structure editing makes this constraint easy to cope with.

This paper completes the work on proof-by-pointing by showing how the constraint of structure awareness can be handled without a structure editor. The proof assistant passes structure information along with the text of data to the editor, which converts the structure information into *extents*. These extents make it very easy to interpret the user's mouse interactions to produce path information, which is interpreted by a proof-by-pointing command generator. The structure of the command generator itself also deserves study and we demonstrate its simplicity and extensibility.

We document the extension of a user interface for the LEGO proof assistant [9] with proof-by-pointing. The interface is implemented inside the XEmacs text editor [18]. We use examples of code and output from LEGO to illustrate our work, but we describe the work with extents in terms of an abstract datatype which could easily be implemented in other graphical environments.

The plan of this paper is organised as follows. Section 2 presents the overall organisation of the proof-by-pointing facility, the components it involves, and the way they interact. Section 3 details three aspects of these components: the pretty-printing mechanism used to pass extra information to the text editor, the extents mechanism used in the text-editor to interpret mouse interactions, and the command generator. Section 4 contains concluding remarks.

2 User-Interface Organisation

From the user's point of view, adding proof-by-pointing to the proof environment implies a change in the aspect of this environment and a change in how to use it. In this section, we will first describe the main changes brought to the interface organisation and to the way it is intended to be used. Then we give a broad picture of the various components that make this organisation possible and how they interact.

2.1 Modification to the Work Environment

We will first describe succinctly what we understand as the initial situation, then we will show how this is modified by our work.

2.1.1 Traditional Situation.

As a starting point, we consider working with a text editor that manipulates several independent textual objects, named buffers. One of the textual documents handles the communication with an external process, this buffer will be named a *shell buffer*. In our experiment of course, this shell-buffer will handle the communication between the text editor and the proof engine. This shell-buffer is organised so that it is possible to insert text in this buffer, send text from this buffer to the external process, and record the output of the process in this buffer. In this approach, the shell-buffer contains both the user input and the external process output.

A more elaborate organisation makes it easier to keep track of the commands sent to the engine. This organisation uses an extra buffer, which we shall call the *script* buffer, intended to record only the commands sent by the user to the proof engine. The normal behaviour is that the user edits the commands in this buffer and, with control keys or menus, sends them to the external process. Internally, the relevant text fragment is still copied to the shell-buffer and the output still appears there, so that the user can keep an eye on this buffer to see the computed data or check that things are going well.

In practice, there may be as many script buffers as the user wants, as data moves only from script buffers to the shell-buffer. On the other hand, it is sensible to have only one shell-buffer, which may be created only when processing the first text fragment.

2.1.2 Adding the Proof-by-Pointing Feature.

The main modification brought about by proof-by-pointing is that some of the output from the logical engine is used to produce commands at the mere click of the mouse. This output is displayed in an extra buffer, which is opened if needed and updated every time the logical engine produces relevant data. The relevant data consists of the lists of assumptions and obligations produced when one works with LEGO to perform goal-directed proof. The special buffer created to receive this data contains a buffer we will refer to as the *goals* buffer. A meaningful click of the mouse in this buffer causes a new

command to be inserted in a script buffer and to be executed by the logical engine. In return, this command changes the data displayed in the goals buffer and the whole system gets ready for a new interaction. Complete goal-directed proofs can be done using only mouse interaction. As in other implementations of proof-by-pointing, mouse interaction can also be interleaved with bouts of manual editing performed directly in the script buffer.

2.2 Architecture

The implementation of proof-by-pointing in an environment based on a programmable text editor requires close interaction between the logical engine and the editor. This interaction revolves around several components communicating through procedure calls or streams of characters and reacting to events provoked by user interaction. Some of the components were already in the proof engine and the text editor before starting our experiment and we only adapted them to our purposes. Other components were created specifically for the needs of this experiment. In this section we will review these components and the data structures they communicate to each other.

2.2.1 Static View.

Figure 1 can be used to understand the broad organisation of our proof-by-pointing tool. The horizontal dashed line in the middle represents the border between two processes, so arrows crossing this line represent communication in text form. The other arrows between the rectangular boxes represent function or procedure calls.

On the text editor side (Emacs side), the boxes represent buffers, and the arrows represent data movement from one buffer to the other. The data is not always plain text: it can also be text enriched with *extents*, regions that make it easy to make the text mouse-sensitive and to attach annotations to locations in the text. In our experiment, these extents have been provided for free by the XEmacs editor. We describe their use later in section 3.2.

On the proof checker side (LEGO side), the boxes represent modules of the checker's implementation. The data structures communicated from one module to the other can vary in form: in particular, the communication between the logical engine, the formatter and the proof-by-pointing command generator uses a representation of logical formulas that is provided by the type checker's implementation and that did not need any modification for our experiment.

When designing this organisation, there was a choice of whether to implement the proof-by-pointing command generator in the text editor or on the proof checker's side. As seen in section 3.3, the choice of implementing it on the proof checker's side makes good use of the ML language and its pattern matching facility to provide easy extensibility. On the other hand, it complicates the communication between the proof checker and the text editor.

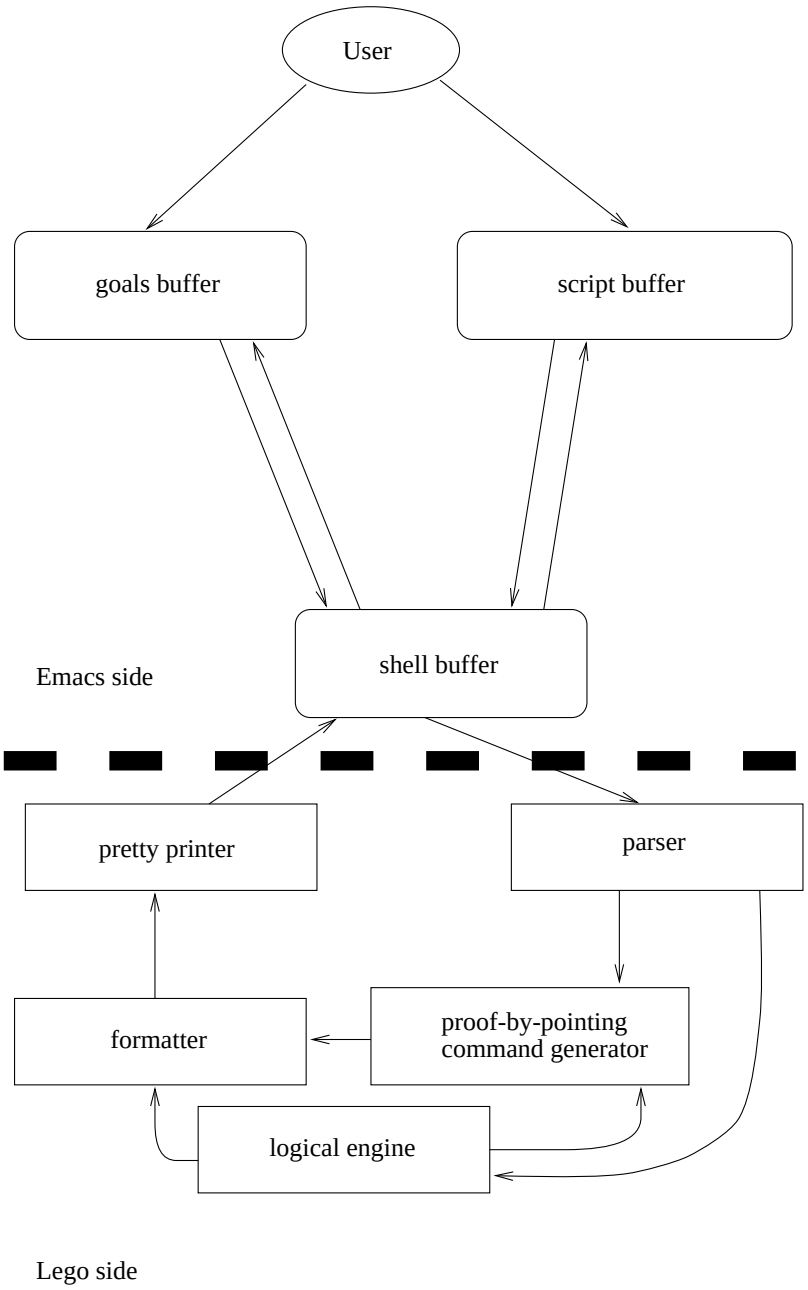


Figure 1: Main architecture of the user-interface

2.2.2 Dynamics

To fully understand the organisation of this experiment, it is also necessary to have a temporal view of its operation, to understand the characteristics of each data type involved, and to understand the conversions that occur in each box. We will give an overview of these conversions by following a complete cycle of the permanent dialog between the two processes.

The typical interaction scenario starts with the user inputting a command in the script buffer of the text editor and requesting that this command be executed by the proof checker, either by choosing an option in a menu or by typing a specific key sequence. The command goes through the shell-buffer to the proof checker's parser, which converts it into a procedure call for the logical engine (note that no modification of the logical engine was necessary for our experiment). In return, the logical engine produces some output. The interesting case for this paper is when the logical engine's output is a new list of assumptions and obligations.

Printing Goals. The logical engine's output goes through the formatter and the pretty printer of the proof checker. These two components are regular components of the LEGO proof checker. In the regular implementation of the proof checker, the pretty printer is a close implementation of the pretty printing algorithm given in [13], and the data type used between these two components is similar to the block datatype described there. The formatter simply produces a block structure which the pretty printer converts into text, taking care of line breaking and indenting in a way that makes the structure easy to comprehend. For our experiment, these two components have to be modified. The block data structure is augmented with a new constructor that makes it possible to print *annotations*. Annotations are text fragments that will be printed by the pretty printer, but whose size will not be taken into account for line breaking and indentation. The intuitive purpose of these annotations is to transfer local information to an external tool, knowing that this local information will be used but not shown to the user. The pretty printer always computes the width of the elements it prints. It is modified to take annotations into account: an annotation is simply treated as a string whose width is zero, but it is always printed. Of course, formatted text containing annotations does not look formatted at all, but it is the receiving tool's job to restore the text's initial elegance by removing the annotations[1].

The formatter is also modified to produce pertinent annotations, which will make it possible for the text editor to render the structure of the logical formulas. In a nutshell, annotations are inserted at the beginning and at the end of each sub-expression of the goals. These annotations contain *path* information that indicates the exact location of the sub-expressions in the goal structure. Later in this paper, we devote a section to this enhancement of the pretty-printer (see section 3.1).

[1]Clearly, one could use such annotations for many purposes, like indicating font or colour changes, thus producing a simple form of *rich text format*.

Last, it is also important to note that the goals are printed between *brackets*. Before starting to print the goals, the proof checker prints a special keyword used solely for this purpose (an opening bracket in some sense). After printing the goals, it also prints another special keyword (the corresponding closing bracket). These brackets will be used to trigger specific actions on the text editor side.

Receiving and Displaying Goals. On the text editor side, the annotated text is received in the shell-buffer. The text editor is organised in such a manner that we can specify that a given procedure will be called every time some data comes in from the proof checker (in XEmacs the procedure only needs to be declared as a `hook`). For the purpose of our experiment, we specify a procedure that will look for the special keywords used as brackets around goals.

When the closing bracket of a pair is recognised, the string the pair delimits is processed to take the annotations into account, remove these annotations, and produce a new kind of text, enriched with *extents*. The pertinent aspects of these extents, provided for free by the XEmacs text editor, are described in section 3.2. The text with extents is then transferred to the goals buffer where it is displayed to the user.

Reacting to User Input. Computation stops there, the next relevant event will come from the user. Most often the event will be generated using one of the other user interface mechanisms, but if the user chooses to proceed using proof-by-pointing, he or she clicks on some expression in the goal buffer. The text editor automatically computes the extent that contains this expression. This extent contains a property which is the path referring to the expression. This path is given in textual form as a sequence of integer numbers. A textual command is constructed with this path and sent directly to the proof checker, via the shell-buffer.

Processing Paths. On the proof checker side, the path information is processed by a specific module, the proof-by-pointing command generator. This module also needs the structured representation of the goal under scrutiny. With these two pieces of information, the proof-by-pointing processor generates plain proof commands for the proof checker in a recursive fashion, using a table of formula patterns and paths. This command generator is described more carefully in section 3.3. The generated commands are just regrouped in a text fragment that is bracketed with special keywords and sent back to the text editor.

Recording and Executing the Commands. On the text editor side, the same procedure that scans the proof checker's output for bracketed goals is also used to scan for the commands generated by the proof-by-pointing processor. These commands are directly inserted in the script buffer and re-sent automatically to the proof checker, which processes them and may produce new goals. The cycle can restart.

Notice that the Proof-by-Pointing tool acts merely as a command generator. Alternatively, one could choose to record the direct Proof-by-Pointing request in the script buffer

and transparently execute the generated low-level command sequence. Our design decision results in more elaborate communication. However, novice users may find it helpful to inspect the output of the command generator. Furthermore, the generated command can later be edited by the user, for instance to adapt generated names to the specific case, or to replace some commands by more efficient ones. This manual post-processing of proof by pointing generated commands could be made easier in a variant where the generated commands are only inserted in the script buffer, but not sent back automatically to the proof checker. This opportunity for manual post-processing would also allow for the command generator to produce incomplete commands.

3 Components

3.1 Producing Annotated Text – Explicit Annotations

Output from the proof system is annotated by the pretty-printer to make information about the internal structure of terms available to the user interface component. Each subexpression in a goal or hypothesis is prefixed with a sequence of integers that represents the path in the abstract syntax tree at which this subterm is rooted. For example, the command

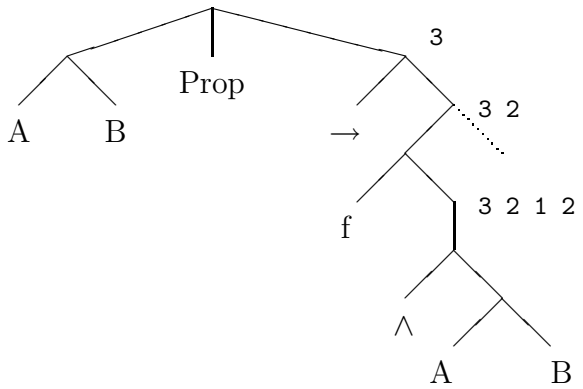
```
Goal {A,B: Prop} (f (A /\ B))  -> (f A /\ f B);
```

causes LEGO to generate the fully annotated[2] response:

```
[ Start of Goals |
  &?0 : [|{[1|[1 1|A],[1 2|B]]:[2|Prop]}[3|[3 2 1|([3 2 1 1|f] [3
2 1 2 1| ([3 2 1 2 1 2 1|A] [3 2 1 2 1 1|∧]
[3 2 1 2 1 2 2|B])])
][3 1|→][3 2 2|([3 2 2 2 1 |[3 2 2 2 1 1|f] [3 2 2 2 1 2 1|A]] [3
 2 2 1|∧] [3 2 2 2 2|[3 2 2 2 2 1|f] [3 2 2 2 2 2 1|B]])]]]
[ End of Goals |
```

There are three different kinds of annotations above: The outermost pair of markers, "`[ Start of Goals |`" and "`[ End of Goals |`", bracket the section of output to be processed for proof-by-pointing. The "`&`" marks the term as a goal or an hypothesis. And within the term itself are the structure annotations. Each annotated subterm has the following format:
`[` *annotation* `|` *term* `]`
The fragment *term* may itself contain annotated subterms. So the output here represents the syntax tree on the following page, in which some example paths have been marked.

[2]The annotations used in practice are selected to be efficiently parsable, and have been replaced here by more readable characters



Note that not all of the nodes in the above tree are annotated in the pretty-printer output – for example, the proof system internally represents a function application as a pair consisting of the function and a list of arguments, but such lists of arguments are never annotated – here the argument list of `/\` is itself a subterm rooted at the path `3 2 1 2 1 2`, which is not present among the annotations. In general, annotations are omitted for subterms that are "mute" with respect to the proof-by-pointing algorithm.

This representation is very flexible: for example the proof engine does not distinguish between infix and prefix operators, conversion being performed by the pretty-printer and parser. Annotating each subterm fully is a simple way to ensure that information about the internal structure of terms is preserved even though there maybe discrepancies between the term structure and its textual representation. In our example, the operator `/\` with the path `3 2 1 2 1 1` appears between its first argument with path `3 2 1 2 1 2 1` and its second argument with path `3 2 1 2 1 2 2`. However, full annotation is very verbose. In order to decrease the volume of data which is generated and parsed, each annotation is preceded by the length of the prefix it shares with its predecessor. Further, spaces are omitted in annotations and the integers which make up the path components are represented by the corresponding ASCII characters, making the assumption that path annotations only contain integers smaller than 256. For example, the above annotation is reduced to:

```
[ Start of Goals |
  &?0 : [0|{[01|[11|A],[12|B]]:[02|Prop]}[03|[121|([31|f] [321|(
[521|A] [51|∧] [522|B])])][11|→][122|([321|[51|f] [521|A]] [31|
∧] [322|[51|f] [521|B]])]]]
[ End of Goals |
```

Although on this small example the value of these optimisations is not wholly convincing, the syntax trees output by the proof checker are typically highly unbalanced, thus with large expressions we achieve significant reductions in output size.

3.1.1 Encoding Annotations

Since the pretty-printing algorithm itself consists of a recursive traversal of the syntax tree, adding path annotations requires only that we keep track of the current path, augmenting it at each recursive call and printing it as a prefix to each subterm of interest. For example, the code

```
fun format_infix name a1 a2 =
  let val sym = infix_sym name
      val arg1 = format a1
      val arg2 = format a2
  in arg1 @ [string sym] @ arg2
  end
```

becomes

```
fun format_infix path name a1 a2 =
  let val sym = infix_sym name
      val arg1 = path_wrap (1::2::path) (format (1::2::path) a1)
      val arg2 = path_wrap (2::2::path) (format (2::2::path) a2)
  in arg1 @ (path_wrap (1::path) [string sym]) @ arg2
  end
```

where

```
path_wrap: int list -> block list -> block list
```

inserts in front of its second parameter an annotation containing the string representation of its first parameter, which is just the path stored in reverse as a list. Path compression occurs only when the annotations are printed out and makes no assumptions about the order of annotations; thus none of the flexibility of fully annotating terms is lost.

3.2 Extents: Annotated Text Intervals

Annotations will be decoded to produce extents. In this section, we describe these extents, the decoding process, and their use to produce proof-by-pointing actions.

Intuitively, an extent is a text interval which is decorated with properties. It is possible to view extents as an abstract data type [3] with the following primitives[4]:

[3] XEmacs provides such a datatype for free, but a concrete implementation with overlays in FSF Emacs or similar mechanisms in systems such as Tcl/Tk would be straightforward

[4] we use the syntax of ML modules, even though XEmacs does not use such modules.

```
signature EXTENT =
  sig
    type extent
    val make_extent          : int → int → buffer → extent
    val set_extent_property : extent → property → value → extent
    val extent_property      : extent → property → value
    val extent_at            : int → buffer → property → extent
  end
```

The constructor `make_extent` f t b generates an extent associated with the buffer b and the text interval given by the start position f and the end position t. The method `set_extent_property` e k v adds a property to the extent e, associating the value v to the property key k. `extent_property` e k retrieves the value associated to the property k for the extent e. Thus, the following equality is true:

$$\texttt{extent_property}\ (\texttt{set_extent_property}\, e\ k\ v)\ k = v\ \ .$$

Example 3.1 (`set_extent_property`) Assume that the extent e ranges over a full term. We wish to record the nature of a term in the property *'pbp-top-element*:

- `set_extent_property` e *'pbp-top-element* $[H, id]$ records that the extent e's associated text interval corresponds to the assumption id
- `set_extent_property` e *'pbp-top-element* $[G, ?7]$ records that the extent e's associated text interval corresponds to the proof obligation identified by the existential variable ?7.

Finally, `extent_at` p b k extracts the smallest extent containing the position p in buffer b and having a property for the key k (in our case, if two extents overlap, then one is included in the other).

XEmacs provides many other capabilities around this concept of extent, but for the purpose of this paper the ones described above are enough. Still, for the user, pointing at a particular position should leave no doubt which subterm corresponds to the position of the mouse, and an ergonomic implementation should foster this link. In our XEmacs implementation, we were able to exploit additional support for the extents property *mouse-face*. It records a font and colour used to highlight the extent when the mouse moves over it, thus making the logical formula's structure strikingly apparent.

3.2.1 Decoding Annotations

When an annotated string representing the proof state is output by the proof-checker, it is parsed by the interface to create the extents for selection by the user. Because we have chosen XEmacs as the setting for our interface and the XEmacs Lisp interpreter is not particularly fast, parsing can be an expensive operation. In view of this, and the fact that

the user may choose some other input method, we do minimal work before presenting the user with output, leaving the rest until a selection has actually been made.

First, the *input* string is stripped of annotations to obtain the *output* string, which is inserted into a clean selection buffer. For each annotation in the input string there is some corresponding point in the output string, specifically the point after all the non-annotation characters which occur in the input string before this annotation. We will call the position of an annotation in the input string its *input position*, and the corresponding position in the output string its *output position*.

A second pass is made over the input string to produce two kinds of extents. The first kind carries a property *pbp-top-element* and is intended to receive the whole text of an assumption or a goal. The property value indicates whether it is a goal or an hypothesis, and the hypothesis name or goal number. The second kind of extent carries a property *pbp-element* and is intended to receive the text of any subterm. The value of this property is the point in the input string at which the annotation of the extent starts. The two kinds of extents are distinguished by the property name.

- Ampersand characters "`&`" are used to distinguish the beginning of goals or assumption. The character `&` has been chosen because it is known to never appear in the regular output. When encountering this character, two cases may arise.

  1. If this is the first `&` encountered, its output position is recorded, together with whether it represents the start of a goal (recognised by a question mark followed by a number) or an hypothesis (recognised by an identifier), and the goal number or hypothesis name.
  2. If an ampersand has been encountered previously, an extent is constructed with the previously recorded position as start position and the current output position as end position, and decorated with a property *pbp-top-element* and the recorded type and name. The new position, type, and number or name are recorded.

- if a `[` is encountered, the input and output positions are placed on a stack. The text is then skipped until a `|` is found.

- if a `]` is encountered, the top input and output positions are popped off the stack. The popped output position is used together with the current output position to construct an extent, which is then decorated with the property *pbp-element* and the popped input position.

- When encountering the closing bracket "`[ End of Goals |`", an extent is constructed with the output position recorded at the last ampersand, and decorated with a property *pbp-top-element* and the type and goal number or name that were also recorded.

The input string is retained as part of the state of the selection buffer. Notice that for any extent with the property *pbp-element*, the input position of the beginning of its annotation

is sufficient to reconstruct its entire path: because of path compression this not quite trivial, but it is straightforward.

3.2.2 Decoding user input

When the user points at a sub-expression in the goals buffer, a procedure is automatically called to interpret this gesture. This procedure uses the primitive `extent_at` to find the two extents e_{top} and e_{sub} with properties *top-pbp-element* and *pbp-element* corresponding to this position. Depending on whether the type of e_{top} is *goal* or *hypothesis*, the goal buffer sends to the shell-buffer a `pbphyp` command or a `pbp` command. In both cases, the command receives as arguments the goal number or hypothesis name and the path computed using the *pbp-element* property of e_{sub}. This command is then automatically forwarded to the proof engine.

3.3 Converting Paths into Command Sequences

The proof-by-pointing command generator uses a representation of the goal or assumption under scrutiny and the path given by the mouse interaction to produce a list of regular commands. It uses a surprisingly simple core function named `pbp`, an extensible collection of rules, which are themselves `ML` functions, and two wrapper functions, which decode the command received from the text editor and translate it into calls to the core function. In this section we describe successively the core function, the rules, and the wrapping functions.

3.3.1 The Infrastructure

The heart of the engine is the function `pbp`, with the following type:

$$\texttt{pbp: string option} \rightarrow \texttt{prCnstr} \rightarrow \texttt{int list} \rightarrow \texttt{string list}$$

The first argument, an optional string, indicates the name of the hypothesis in which the algorithm is working. If this name is absent, then the algorithm is working in a goal[5]. The second argument is the term structure of the statement of the goal or premise on which the algorithm is working. The third argument is the path received from the text editor.

The algorithm works by successively trying all the rules found in a mutable repository, `PBP_RULES`. Each rule is represented by a function (`f` in the code below). When a rule succeeds, it returns a list of strings, representing the generated commands. Note that the function `pbp` itself is passed as an argument to each rule, so that the recursive nature of the proof-by-pointing algorithm is implemented by the possibility that rules call this function recursively on a subterm. In this respect, the proof-by-pointing algorithm is implemented using a *continuation passing style*.

[5]In this presentation we assume that the proof process automatically focuses on a specific goal

```
fun pbp (h:string option) (t:prCnstr) (p:int list) =
  let fun try_all_rules rl =
      case rl of ((_,f)::tl) =>
        (case (f h t p pbp) of SOME(command_list) => command_list
           | NONE => (try_all_rules tl))
      | [] => []
  in
      try_all_rules (!PBP_RULES)
  end;
```

The property of PBP_RULES to be a mutable reference cell is not essential in the pbp core function. However, it makes it easy to add new proof-by-pointing rules dynamically. Actually, the elements of the PBP_RULES list are not only functions, but pairs of a string and a function. The string is used as a rule name, and makes it possible to remove proof-by-pointing rules by referring to their name. The names compensate for the fact that the rule values cannot be inspected, because they are plain ML functional values.

3.3.2 Implementing Proof-by-pointing Rules

To construct pbp rules, we need the proof engine to provide the following primitives:

```
signature PBPINPUT =
    sig
        val mkNameLoc      : string → string
        val initNameLoc    : unit → unit
        val getCurrentGoal : unit → int * prCnstr
        val getAssumption  : string → prCnstr
    end
```

The name generator mkNameLoc returns a fresh identifier such that mkNameLoc$s$ = s provided s is already fresh. The context of local identifiers i.e., those introduced by mkNameLoc, can be initialised by initNameLoc. The function getCurrentGoal returns the current goal's tag and type. Finally, getAssumption$s$ retrieves the type of the assumption s.

Proof-by-pointing rules have approximately the same type as the core function, except that they have an extra argument, which they use as a continuation, and their returned value is optional. Thus, their type is exactly the following one.

```
(* hypothesis   *) string option →
(* subterm      *) prCnstr →
(* path         *) int list →
(* continuation *) (string option → prCnstr → int list →
                                  string list)
(* command      *)  → string list option
```

The structure of proof-by-pointing rules is practically always the same. They must perform the following operations:

1. Check that the formula and the path correspond to a case where this function should apply. In case of failure, return `NONE`.
2. Determine the sub-formula and the sub-path needed for a recursive call of the proof-by-pointing algorithm.
3. Compute the command corresponding to this rule.
4. Call the `pbp` function, with the right hypothesis name (when necessary), the sub-formula and the sub-path.
5. Return the concatenation of the command computed at step 3 and the result of the recursive call.

Steps 1 and 2 are easily performed thanks to the pattern matching facilities of `ML`.

Of course there may be no recursive call. For instance the following rule describes a sensible behaviour for the empty path:

Example 3.2 (Axiom) We wish to discharge the sequent $A, \Gamma \vdash A$ whenever the goal A has been selected.

```
fun pbp_catch_empty_path NONE _ [] _  = SOME ["Try Assumption"]
  | pbp_catch_empty_path _    _ _  _  = NONE
```

Restricting the specification to the empty path guarantees that the complete goal has been selected and not merely a subterm. The second line of this function ensures that the rule fails cleanly if the pattern is not adapted.

When there is a recursive call, the rule must be designed so that the arguments are coherent: the sub-path should be valid in the subterm. This is usually easy to check, under the assumption that the path and term given as arguments are already coherent. Also, to ensure termination, either the sub-path or the subterm should be a proper sub-component of the corresponding initial value. These design issues have been taken care of in the following example:

Example 3.3 (Introduction of Conjunction) We wish to implement the rule

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \boxed{\mathrm{B}}}{\Gamma \vdash A \wedge \boxed{\mathrm{B}}} \ .$$

Notice that the LEGO system supports a corresponding refinement by

$$\Gamma \vdash A \wedge B \xrightarrow{\texttt{Refine pair}} \Gamma \vdash A \quad (\Gamma \vdash B) \xrightarrow{\texttt{Next}+1} \Gamma \vdash B \quad (\Gamma \vdash A)$$

A rule using this refinement is implemented as follows:

```
fun pbp_andr2 NONE (PrApp (PrRef "and",[_,(B,_)]))
              (2::2::path_tail) continuation
              = SOME (["Refine pair", "Next +1"] @
                      (continuation NONE B path_tail))
  | pbp_andr2 _ _ _ _ = NONE
```

The `prCnstr` constructor `PrApp` expects a list of arguments, each tagged with display switches. Those switches are redundant for the purpose of implementing Pbp rules. This use of a datatype initially intended for other purposes makes the rules a bit less readable.

In some cases, new hypotheses may be generated and the recursive call may be performed on one of these new hypotheses. It is then necessary to create a new name, use this name in the generated command, and pass this name on to the recursive call.

Example 3.4 (Elimination of Implication) Consider the rule

$$\frac{A \Rightarrow B, \Gamma \vdash A \quad \boxed{\mathrm{B}}, A \Rightarrow B, \Gamma \vdash C}{A \Rightarrow \boxed{\mathrm{B}}, \Gamma \vdash C} \ .$$

Let H be a proof of $A \Rightarrow B$ available in the context Δ. Using LEGO, the above refinement can be accomplished by

$$\Delta \vdash C \xrightarrow{\texttt{impE}\,H} \Delta \vdash A \quad (\Delta \vdash B \Rightarrow C) \xrightarrow{\texttt{intros}+1\ b} B, \Delta \vdash C \quad (\Delta \vdash A)$$

where b with type B is a fresh identifier generated by `mkNameLoc`:

```
fun pbp_impr2 (SOME H) (PrBind([], (Pi,_), _, B))
    (2::2::path_tail) continuation
    = let val b = mkNameLoc "b"
      in  SOME (["impE " ^ H,"intros +1 " ^ b] @
                continuation (SOME b) B path_tail)
      end
  | pbp_impr2 _ _ _ _ = NONE
```

The very simple nature of this organisation could also allow *point-and-shoot* extensions as described in [3].

Example 3.5 (Point-and-Rewrite) Consider the rule

$$\frac{a = b, \Gamma \vdash C[b/a]}{\boxed{\mathrm{a}} = b, \Gamma \vdash C}$$

where the complete term a has been selected and not merely a subterm. Let eq be a proof of $a = b$. LEGO supports a corresponding refinement by `Qrepl`eq. This leads us to the following implementation:

```
fun pbp_repl (SOME eq) (PrApp (PrRef "Eq",_)) [2,1] _
                    = SOME ["Qrepl "^eq]
  | pbp_repl _ _ _ _ = NONE;
```

3.3.3 Wrapping functions

Both `pbphyptop` and `pbptop` are wrappers of the core function `pbp`. They use the primitive `initNameLoc` to initialise the local namespace. In addition, `pbphyptop` *id l* invokes `pbp` with (`SOME` *id*), the type of *id*, computed with the primitive `getAssumption`, and *l*. Similarly, `pbptop` *g l* invokes `pbp` with `NONE`, the type of the goal *g*, computed with the primitive `getCurrentGoal`, and *l*.

4 Conclusion

This paper shows that the proof-by-pointing facility can be integrated into an Emacs-based environment at very low cost. The developments needed in the proof engine are the modification of the pretty-printer and formatter and the implementation of the proof-by-pointing command generator. The developments needed in the Emacs text editor are the decoder that will interpret structure information, and hook functions that will update the goal buffer's contents and interpret mouse interaction. For a prototype implementation, the whole work was done in one week during the stay of one of the authors at Edinburgh in the fall of 1996. This is the main strength of the work described here: proof-by-pointing can smoothly be integrated into a wide variety of proof environments.

This work also shows a potential weakness. If proof-by-pointing rules are represented solely by ML functions, it is not possible to view these rules and to edit them. Ideally the rule system would be defined in an extensible format and implemented with a generic engine, an approach which has been taken in CtCoq [1]. The omission of such a facility weakens our claim of easy extensibility: as the system stands, it can easily be extended but the person willing to do so had better be an ML programmer and make only a few mistakes. Debugging faulty proof-by-pointing rules still remains an unsupported task.

Two questions linger. First, how easy would it be to implement this in a framework where the proof engine uses a pretty printer that is itself extensible? Solving this question would make it easier to implement this facility for systems that tend to have a "logical framework" approach like Coq or Isabelle. Secondly, would it be easy to implement this in a text editor which lacks extents? The answer seems to be that it would: rather than relying on a concrete implementation on the text editor side, extents could be administered by maintaining a table of intervals and properties in the proof engine. In this case the text-editor would only need to send back the character position of the mouse selection, instead of a path at every mouse interaction in the goals buffer. The disadvantage of this approach would be that the information provided by highlighting selectable extents as the mouse moves over the selection buffer would no longer be available to the user. In any case, our experiment shows that a structure editor is not a necessary part of an interface which supports proof-by-pointing.

The notion of extents has proved its power and it is also interesting to understand how they can help with other issues of proof environments. For instance [3] introduces a notion of script management. We have already started to experiment with implementation of

similar facilities where extents are used to manage the basic elements of scripts: individual commands.

5 Acknowledgements

Special thanks go to Lena Magnusson who experimented with extents for the Emacs mode for the Alf [10] proof system during the spring of 1996 at INRIA-Sophia Antipolis and shared conclusions on these experiments with Yves Bertot.

Helpful comments on a draft of this paper were also received from James McKinna, Healfdene Goguen, and the referees for TYPES '96.

The collaboration between Edinburgh and Sophia-Antipolis has been supported by the EC's Esprit Working Group *TYPES*. Furthermore, T.K. and D.S. are grateful for being funded by the EC's TMR programme and EPSRC, respectively.

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] Richard Bornat and Bernard Sufrin. Jape: a literal, lightweight, interactive proof assistant. Technical Report 641, Queen Mary and Westfield College, University of London, 1994.

[5] Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly & Associates, Inc., 1991.

[6] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[7] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[8] Masami Hagiya and Hiroshi Kakumo. Proving as editing. In Nicholas A. Merriam, editor, *User Interfaces for Theorem Provers (UITP 96)*, pages 35–42. Department of

Computer Science, University of York, July 1996. `http://dcpu1.cs.york.ac.uk:6666/~nam/uitp/proceedings.html`.

[9] The LEGO World Wide Web page. `http://www.dcs.ed.ac.uk/home/lego`.

[10] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.

[11] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.

[12] S. Owre, S Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided verification*, volume 1102 of *Lecture Notes in Computer Science*, July 1996.

[13] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[14] Brian Ritchie. *The design and implementation of an interactive proof editor*. PhD thesis, University of Edinburgh, 1988.

[15] Donald Syme. A new interface for HOL - ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, September 1995.

[16] 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.

[17] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5):120–129, 1992. 5th Symposium on Software Development Environments.

[18] The XEmacs World Wide Web page. `http://www.xemacs.org`.