May 1986

Report No. STAN-CS-86-1109

A Proof Editor for Propositional Temporal Logic

by

Ross Casley

Department of Computer Science

Stanford University
Stanford, CA 94305

A PROOF EDITOR FOR PROPOSITIONAL TEMPORAL LOGIC

Ross Casley

Computer Science Department
Stanford University

1. Introduction

This report describes PTL, a program to assist in constructing proofs in propositional logic extended by the operators □ ("always"), ◇ ("eventually") and ○ ("at the next step"). This is called propositional temporal logic and is one of two systems of logic presented by Abadi and Manna in [1].

PTL is neither a proof generator nor a proof checker. Instead, it is a proof editor. It provides convenient commands to manipulate logical formulas in order to ease the task of constructing valid proofs.

PTL is written in Zetalisp and runs on the Symbolics 3600 workstation.[1]

PTL is not a stand-alone program. It is a number of additional commands that extend the 3600 editor, Zmacs. All of the deduction rules of propositional temporal logic are implemented.

1.1 Outline of this Report

Chapter 2 begins with an example of proof construction using PTL. The second part of this chapter is a user's manual describing all the commands available.

Chapter 3 is an overview of PTL internals. In it I show that the code consists of four relatively independent sections. Chapters 4 to 7 concentrate on each of these four sections in turn. In these chapters I discuss the reasons for choosing particular algorithms, as well as the algorithms themselves.

PTL uses undocumented features of Zmacs. These features are described in Appendix A.

This research was supported in part by the National Science Foundation under grant DCR-84-13230.

[1] Zetalisp, Symbolics and Symbolics 3600 are trademarks of Symbolics, Inc.

2. PTL Users Manual

I assume that the reader has a basic knowledge of Zmacs, including entering and saving text, and using the mouse to move the cursor and mark text. However, I will assume as little knowledge as possible about other parts of the Symbolics environment. Knowledge of Abadi and Manna's proof system is also assumed.

2.1 Terminology

In this manual the term *formula* is applied to any PTL formula, whether it occurs as a single line of the proof, or as a part of a larger formula. The term *whole* formula is used to restrict meaning to ii. formula that forms a line of the proof.

The term *conjunction* always refers to a formula in which two or more subformulas are joined by the A operator. The subformulas are called *conjuncts.*

Similarly, a *disjunction* is a formula consis ting of two or more *disjuncts* joined by the v operator.

2.2 Example of Using PTL

The best introduction to PTL is by example. Here I give instructions for constructing a proof of $\bigcirc p \supset \Diamond p$. The reader is encouraged to follow the instructions while running PTL. Even if this is not possible, do not skip this sec tion; the basic operations used to run PTL are not repeated elsewhere. More detail on the PTL commands is given later in this chapter.

Loading the PTL System

If PTL is not already loaded, give the lisp command (load `"c : >ross>ptl>system-def"`). Lisp will respond with a question ending: load all 5 of them? `(Y , N, S)`. You should type Y. (If this question did not appear, PTL was already loaded and you can continue.)

Creating a New File

Create a new file to hold the proof. The file can be given any name, but the extension should be . `ptl`. This extension ensures that Zmacs will au tomatically edi t the file in PTL mode. The special PTL commands arc only available for files edi ted in PTL mode.

· Ill the first line of this file we will simply describe the deduction to follow. Of course, PTL does not require that this be done, it is just, our choice to do so. So we want to insert a line containing something like "Here is il. proof of $\bigcirc p \supset \Diamond p$." This brings us to the first problem, how to type the logical symbols.

Typing Logical Operators

The three special keys marked with a square, circle and triangle can be used to enter the modal operators □, ◯ and ◇ respectively. The other logical symbols are standard on the 3600 keyboard, and are typed using the "symbol" key. Unfortunately, the symbols do not appear on the keyboard, but their position can be discovered by typing symbol-Help. Symbol-Q is the A-sign, symbol-W is V, symbol-K is → and symbol-Y i s ⊃. (3 and –› are equivalent ways to type "implies".) The carat (shift-6) is *not* the same cha rac ter as the and-sign, A.

Now type the first few lines of the file so that it appears as follows. (Note: the first, line displayed is an attribute list for the file. It was created using the Zmacs commands, PTL Mode and Set Package. This line may be omitted if you wish.)

```
-*- Mode:  PTL; Package:  USER -*-
First a proof that ◯p ⊃ ◇p.  Proof is by contradiction
```

Text such as this may appear anywhere in the file, and does not affect the deduction.

Typing a Formula

Now type the premise of the deduction. Since this is a proof by contradiction, the premise is $\neg(\bigcirc p \supset \Diamond p)$.

Formulas that are part of the proof are distinguished from plain text by appearing on a new line beginning with a *line number*, which is an integer followed by a colon. The line number can be entered by typing it, or it can be generated automatically using the "Super-L" command.

Line numbers need not be unique. However proofs containing duplicated line numbers are difficult to read.

After entering the formula the file should contain:

```
-*- Mode:  PTL; Package:  USER -*-
First a proof that ◯p ⊃ ◇p.  Proof is by contradiction
1:  ¬(◯p ⊃ ◇p)
```

Performing a Deduction

There are now several ways to proceed. The strategy taken here is first to distribute negations, then apply the □ rule to obtain an explicit contradiction.

The PTL deduction commands are invoked by giving a command which specifies the deduction rule to be applied. The command for distributing negations is "Super-N". Type Super-N now.

PTL is now waiting for you to choose a formula using the mouse. The method used to select premises is common to all rules, and is worth explaining in some detail here.

Notice that the mouse cursor changes to a thick vertical arrow, and instead of pointing to individual characters on the screen, the mouse points to subformulas occurring in the proof. A hollow box always surrounds the formula to which the mouse is pointing. Clicking a mouse button selects the outlined subformula.

Move the mouse so that it points to the formula in line 1. Be careful not to click one of the mouse buttons.

PTL always assumes that the mouse points to the smallest possible formula surrounding the mouse cursor. In the example formula, pointing the mouse cursor to the ◇ sign will outline the formula $\Diamond p$; pointing to the ⊃ sign will outline the formula $\bigcirc p \supset \Diamond p$. In general, a subformula can be outlined by pointing to its logical operator.

When pointing to a formula, each of the three mouse buttons will have a different effect. A [right] click always aborts the command. The effect of [left] and [middle] clicks depend on the command you gave. For most commands the middle button aborts the command, and the left button performs the requested deduction. The inverse-video information line at the bottom of the screen always lists the meaning of each mouse button. Typing anything besides one of the mouse

clicks will abort the command. The unexpected input is put back onto Zmacs' input stream to be interpreted as a command.

To complete this deduction, outline the whole formula $\neg(\bigcirc p \supset \Diamond p)$, by pointing to the $\neg$ operator. Click [left] t0 complete the deduction. PTL inserts the conclusion at the end of the buffer, so the file now contains:

```
-*- Mode:  PTL; Package:  USER -*-
First a proof that ○p ⊃ ◇p.  Proof is by contradiction
1:  ¬(○p ⊃ ◇p)
2 :  ○p ∧ □¬p                                  ; By distributing ¬ in 1
```

The conclusion is written with a new 'line number, and with a comment explaining how the line was derived.

Comments are not restricted to formulas written by PTL. **You** can place comments on any formula by typing a semicolon before the comment.

Expanding the Modal Operator

The next step in this strategy is to apply the • I rule to the subformula $\Box\neg p$ of line 2. The $\Box$ rule is invoked by the "Super-E" command. Super-E invokes the ❑ rule or the $\Diamond$ rule, depending on the main operator of the premise chosen for it. Most, PTL commands invoke one of several rules, depending on the premises chosen.

The Super-E command is invoked in exactly the same way as the Super-N command. In fact we must invoke this rule twice, and the file finally contains:

```
-*- Mode:  PTL; Package:  USER -*-
First a proof that ○p ⊃ ◇p.  Proof is by contradiction
1:  ¬(○p ⊃ ◇p)
2 :  ○p ∧ □¬p                                  ; By distributing ¬ in 1
3: ○p ∧ (¬p ∧ ○□¬p)                            ; By the □ rule from 2
4: ○p ∧ (¬p ∧ ○(¬p ∧ ○□¬p))                    ; By the □ rule from 3
```

Completing the Proof

By now the contradiction should be obvious, we have both $\bigcirc p$ and $\bigcirc\neg p$ occurring (implicitly) in line 4. This suggests that the basic resolution rule should be used to complete the proof. There are two ways to invoke the basic rule. The one used here is the "Super-R" command. The user must choose two premises for this command, both occurrences of the same formula. The command works out a reasonable way to apply the resolution rule so that *true* is substituted for one occurrence and *false* for the other.

When a command uses two (or more) premises, clicking [left] selects the first premise, but no deduction is performed. The hollow box surrounding the first premise remains even when the mouse is moved. A second box surrounds the formula to which the mouse points. Both boxes disappear after the second premise is selected, and the deduction made.

When you click [left] the box surrounding the first premise may disappear. This happens because it is being surrounded twice — once by the blinker that follows the mouse around, and a second time by the blinker that shows it has been selected. These two blinkers cancel each other.

Simply iiiove the mouse, and both blinkers will be visible. For the same reason, you may see unusual outlining whenever two blinkers overlap on the screen.

The Super-R command is unusual in that it generates two lines in the proof. The first new line is the result of the basic rule, the second line simplifies the first line.

After typing Super-R, the user here selects as first premise the leftmost occurrence of p. The second premise is the third occurrence of p, which appears as . . . A ◯(¬p . . . . Do this now, and the following addi tional lines will appear.

```
5: ◯p ∧ (¬p ∧ ◯(¬p ∧ ◯□¬p)) ∧                    ; By resolution from 4
        ((¬p ∧ ◯(¬TRUE ∧ ◯□¬p)) ∨ ◯FALSE)
6:  FALSE                                        ; By simplifying
```

This completes the proof, and illustrates an important point. Sometimes a formula is too long to fit comfortably on a single line. It can be continued onto another line by splitting it So that one of the connectives (A, V, ⊃, or →) appears at the end of the first line (except for comments), or is the first non-blank character of the continuation line. Formulas may be continued onto as many lines as necessary by using this convention on each line.

2.3 Details of Selecting Forrnulas

Several situations did not arise in the example given, bu t need to be mentioned.

In a conjunction, such as A A B A C, pointing the mouse to A will outline the A, pointing to either of the A signs outlines the whole conjunction, A **A** B **A** C. There is no way to outline just a piece of the formula, say A **A** B. If this is necessary for the proof, first use the weakening rule described below to deduce a new formula with A **A** B replacing A **A** B **A** C. Then use the new formula in it premise. Similar comments apply to disjunctions.

111 a long proof, two premises may be SO far apart that they cannot appear on the same screen. But typing a scrolling command such as Control- V aborts the deduction command. The solution is to use the mouse scrolling commands, invoked by moving the mouse to the left border of the Zmacs screen. Scrolling this way does not abort a deduction command.

2.4 How and When Formulas are Parsed

Before any formula can be used as the premise of a deduction, PTL must parse it to determine the position of each of its subformulas. Every formula in a file is parsed when the file is read. (This will cause a noticeable delay if existing files with many formulas are read.) New or altered formulas are parsed when any of the commands initiating a deduction are issued.

The parser is normally case sensitive, so `p` and `P` are distinct formulas. The constant true and false formulas are exceptions to this rule. Any formula with the names `true` and `f` also, no matter how capitalized, will be equivalent to TRUE and FALSE respectively.

Parsing is abandoned if a syntax error is found. The cursor will be positioned at the point where the error was found, and an error message will be given iii the typo-iii window at the bottom of the screen. Parsing may be resumed by issuing a deduction command. The parser may also be invoked using the "Parse Buffer" (super-P) command. This is useful if you expect to find further errors in the text.

Unary operators are given the highest priority, followed by **A**, then **V**, then ⊃. That is, the formula ¬A A B **V** C ⊃ D is interpreted as (((¬A) **A** B) **V** C) ⊃ D. This priority is accepted on

input, but when PTL prints out formulas, it inserts additional parentheses so that the meaning is clear. The A and V operators are treated as variadic, not binary, so additional parentheses are never needed within conjunctions or disjunctions.

2.5 Dual Rules

Each of the deduction rules of propositional temporal logic has a corresponding *dual rule*. For example, the weakening rule allows you to replace a conjunction with positive polarity by one of its conjuncts. The dual of weakening allows you to replace a *disjunction* having *negative* polarity by one of its *disjuncts*. In general the dual rule can be stated by taking the statement of a rule and interchanging "positive polarity" and "negative polarity", "A" and "V", "0" and "◇", and *"true"* and "*false*".

A proof containing dual rules can always be replaced by a proof using only standard rules, but the dual rule proof is sometimes shorter and more clear.

PTL implements duals of all the deduction rules. However, some of the rules are unchanged when dualized, and the • I and ◇ rules are already duals. A dual rule is applied using the same command as the standard rule. For example, the resolution command applied to a conjunction with positive polarity uses the standard rule. The same command applied to a disjunction with negative polarity uses the dual of resolution.

2.6 PTL Commands

Simplify Formula (Super-S)

This command replaces a subformula of the existing formula by an equivalent simpler formula. To simplify the whole formula, simply select the whole formula.

The simplifications used are

- occurrences of *true* and *false* are removed, replacing the subformula containing them by an equivalent formula. For example, A A B A $TRUE$ will become A A B. There are similar replacements for other operators,
- double negations are removed,
- nested conjunctions and nested disjunctions are flattened. For example, $A \wedge (B \wedge C)$ becomes $A \wedge B \wedge C$.

If no simplification can be made, no new line is written to the file.

Weaken Formula (Super-W)

This command replaces a conjunction by a subset of its conjuncts. A [left] click and a [middle] click have different effects. [Left] on a conjunct replaces the whole conjunction by that formula. [Middle] allows you to select more than one of the conjuncts to appear in the final formula. When you click [middle] the conjunct will be outlined, but you will still be able to click either [middle] or [left] on other conjuncts. When you eventually click [left] a new formula containing only the selected conjuncts will be deduced.

The dual of the weakening rule will be used the selected formulas are disjuncts instead of conjuncts. The disjunction must have negative polarity.

After clicking [middle] on a number of conjuncts, you can click [left] while not pointing to any subformula to complete the weaken command without adding any further conjuncts.

Distribute Negations (Super-N)

You must select a formula whose outermost operator is a $\neg$ or $\supset$.

If the operator is $\neg$ de Morgan's laws and negation rules for modal operators will be applied to produce an equivalent formula with negations appearing only at the lowest level. That is, negations will only be applied to propositional variables.

If the operator is $\supset$, the formula is replaced by the corresponding disjunction then the negation rules are applied repeatedly to the result. For example, selecting $\neg A \supset B$ will give the conclusion $A \vee B$.

A simpler deduction is performed if this command is preceded by a numeric argument (that is, by preceding the command by "Control-l" 'or "Control-IT"). For negations, the negation will only be pushed down one level instead of being applied recursively. Implications will be replaced by the equivalent disjunction, without any further application of the negation rules.

Manual Resolution (Super-Q)

This command implements the basic resolu tion rule or its dual.

Resolution is presented symbolically in [1] as follows:

$$R[A\langle u\rangle, B\langle u\rangle] \mapsto A\langle true\rangle \vee B\langle false\rangle.$$

A badi and Manna allow several occurrences of the formula u to be replaced simultaneously, but in PTL only one occurrence in each of A and B may be replaced.

[There is a danger in doing this. Similar restrictions in other logical systems destroy the system's completeness. While we are not much concerned with completeness here the user could always type the required conclusion without using any deduction commands at all it is comforting to know that this system is still complete. To see this, note that the completeness proof in [l] uses only the restricted rule. This fact was pointed out by Martin Abadi.]

The user must choose four formulas. First choose two formulas as the conjuncts A and B. They may be chosen in any order. You may choose two conjuncts within the same conjunction, or you can choose two whole formulas to be A and B.

After choosing A and B, the blinker that outlines the formula you are pointing to will change to a solid blinker. The next two formulas chosen are the occurrences of u those formulas within A and B that are to be replaced by *true* and *false*. The first formula chosen will be replaced by *true*, the second by *false*. The deduced formula will be added at the end of the buffer.

A pplying this rule is fairly tedious. The nex t command described is an attempt to simplify this task.

Automatic Resolution (Super-R)

This command is an alternative to the resolution command of the previous section. Instead of choosing four formulas, you are only required to choose two. These two formulas must both be occurrences of the same formula, and they are taken as the inner formulas of the deduction (called u in the previous section).

PTL attempts to guess what A and B must be to surround these formulas. If the two inner formulas are within the same line, then a formula that contains both the selected formulas is found.

This must be a conjunction, and the conjuncts containing the selected formulas are taken as A and B. If the selected formulas are in different lines, then those lines are taken as A and B.

The two selected formulas must have different polarities. The formula having positive polarity will be replaced by j&e, and the other will be replaced by *true* in the conclusion.

This is often the deduction that is desired. For example, selecting the two occurrences of P in P and $(P \supset Q)$ will result in the conclusion Q. Also, sclccting the two occurrences of a in $a \wedge x \wedge \neg a$ will result in *false*. There is obviously no guarantee that this is always a useful deduction.

This command adds two formulas at the end of the buffer. The first is the resolvent. It will be followed immediately by its simplification.

The dual resolution rule is also implemented. The formula that has negative polarity is still rcplaccd by *true* and the formula with negative polarity by ***false.***

Binary Modality Rules (Super-M).

The six binary modality rules ($\Box\Box$, $\Box\Diamond$, $\Diamond\Diamond$, $\bigcirc\bigcirc$, $\Box\bigcirc$ and $\Diamond\bigcirc$) and their duals are all implemented by a single command.

The rule to apply is selected depending on the modal operators appearing in the chosen formulas. For all rules except ❑ IO the order in which the formulas to be resolved upon are chosen is almost irrelevant to the conclusion drawn. (The $\Diamond\Diamond$ and $\bigcirc\bigcirc$ rules will have different, but equivalent, conclusions if their premises are chosen in a different order.) For the $\Box\Box$ rule, if the first choice is $\Box u$ and the second is $\Box v$, the conclusion is ❑ $_{(CITL \wedge v)}$.

Unary Modality Rules (Super-E)

The two unary modality rules ($\Box$ and $\Diamond$) are also implemented by a single command. (The key binding can be remembered by mentally associating super-E with "expand modal operator".) The appropriate rule is chosen according to the operator of the selected formula.

These two rules are duals of one another.

The Induction Rule (Super-I)

The induction rule or its dual is applied by this command. The standard rule states that if $\neg(w \wedge u)$ can be proven, then

$$R[w, \Diamond u] \mapsto \Diamond\left(\neg u \wedge \bigcirc(u \wedge \neg w)\right).$$

There i s no check that the lemma, $\neg(w \wedge u)$, is ever proved. The required lemma is printed in the co mmen t of the conclusion line, unless this deduction is the special case where $w = \neg u$.

The dual of this rule requires the lemma $w \vee u$, and its conclusion is $\Box(u \supset \bigcirc(w \supset u))$. For the special case $w = \neg u$, the conclusion is $\Box(u \supset \bigcirc u)$.

Distribution Rules (Super-D)

Three distribution rules (for distributing **A** over **A**, **V** and $\supset$) and their duals are implemented by this command. The formula to be distributed must be chosen first, and the formula to be distributed over is chosen second. If the second formula is a disjunction of more than two terms, the formula will be distributed to each of the disjuncts. For example, if the first formula is u and the second is $v \vee w \vee x$, the result will be $(u \wedge v) \vee (u \wedge w) \vee (u \wedge x)$.

The dual of distibution distributes **V** over the other operators.

Parse Formulas (Super-P)

Whenever any deduction command is executed, the buffer is first checked for formulas that have changed since they were last parsed. The super-P command allows you to invoke this parsing process without performing any deduction.

Insert Line Number (Super-L)

This inserts the next available line number at the start of the next line. If the cursor is already at the beginning of the line, no new line is created.

Indent for Comment (Control-;)

This is not a PTL command, it is part of standard Zmacs often useful in PTL. It moves the cursor to the comment position on a line. If there is no existing comment, a semicolon is typed.

2.7 Summary of Commands

Key	Command Name	Rules implemented
Super-D	Distribute formula over A, ∨ or ⊃	
Super-E	Expand Modal Operator	□, ◇rules
Super-1	Induction Rule	
Super-L	Insert Line Number	
Super-M	Modal Resolution	□□, □◇, ◇◇, ○○, □○ and ◇○
Super-N	Distribute ¬	distributing ¬
Super-P	Parse Formulas	
Super-Q	Manual Resolution	Basic resolution rule
Super-R	Automatic Resolution	Basic resolution rule simplified
Super-S	Simplify Formula	*true-false* rules
Super-W	Weakening rules	
Circle Key	Type ○	
Square Key	Type □	
Diamond Key	Type ◇	
Control-;	Begin comment, indented	

2.8 Parameters

A number of parameters can be set by the user. Note that all of them are in package **zwei**. These are not of interest to most users, but are listed here for completeness.

***default** ault-ptl-comment-margin* The number of characters to indent a comment when a derived formula is printed. This applies only to formulas printed by PTL. The indentation used by the Control-; command is set by a different Zmacs variable.

ptl-print-names. An association list giving the external forms of the connectives and operators.

trace-parser. For debugging the parser. If non-null, a line is printed for each shift or reduce action of the parser.

default** ault-ptl-f ont-f **ile. The bfd file containing the special font for printing formulas.

2.9 Fonts

If you list a PTL file without using Zmacs, or if you edit a PTL file without using PTL mode, the □ and ◯ symbols will not be displayed correctly. This is because PTL mode uses a special font, which is set up automatically when you enter PTL mode.

3. PTL Internals

In this and the following four chapters I describe the internals of PTL, and discuss why particular algorithms were chosen.

Design Goals

Design choices were made with respect to some specific goals. Ease of use was emphasised, and this led to several early design decisions. Firstly, the Symbolics lisp machine was chosen because it allowed the use of a mouse to select formulas for a deduction. This is a more natural interface than other methods, and is especially necessary in PTL because the formulas being resolved upon can be nested within other formulas. Sccoxrdly, Zmacs was used for the basic user interface, rather than building a more specialized interface, because it immediately gave access to powerful text editing facilities, multiple window capability, easy storage and retrieval of proofs and allowed the user to swap between tasks easily.

A second goal was to make efficient use of space. For example, conclusions should share structure with premises as much as possible. Failing to control space usage leads to more frequent garbage collections, which slows the machine.

PTL Modules

The PTL software can be divided into 4 pieces, which I will call modules, thought they are not modules in the usual sense. Each module maintains its own set of data structures and algorithms, and will be described separately. The modules arc:

1. Functions for manipulating formulas,
2. Command definitions, mouse handler and other interfaces to Zmacs,
3. A parser, and
4. A prettyprinter.

The parser converts formulas into an internal format. The formula manipulation functions use this internal format, both for input of premises and for output of conclusions. The prettyprinter converts internal format back into external format for output into the file being edited.

The Zmacs interface (module 2) controls processing by the other modules.

4. Formula Manipulation

4.1 Data Structures

There are two data types used by the formula manipulation software, *formulas* and *occurrences*. Both data types are also used outside the formula manipulator, to communicate between other parts of the software.

Formulas

Formulas are the simpler of the two structures. A formula is the internal representation of a logical formula. It uses a very common representation for formulas: propositional variables are represented as lisp atoms; compound formulas are represented as lists, with the car of the list being the logical connective, and the cdr being the list, of arguments. For example $A \supset B$ is represented its (IMPLIES A B), $AA\ (B \vee C)$ is represented as (AND A (OR B C)), and so on. The atoms TRUE and FALSE are reserved for use as the constant true and false propositions.

Occurrences of Formulas

In many situations we need to know where a formula occurs within another formula. The data structure called an *occurrence* is used to represent this information. For example, the formula A appears twice in the formula $A \wedge \bigcirc\Diamond A$, and each instance of A would be represented as a different occurrence.

Occurrences are quite complex to define, and some special terms will be needed to describe them. I will use the term *formula* for either the abstract mathematical object, or its representation in lisp. This should not cause any confusion. Every formula is either *atomic* (consists of a single propositional variable), or *compound*. A compound formula is defined by its *connective* and its list of *arguments*. The allowable connectives are AND, OR, IMPLIES, NOT, ALWAYS, EVENTUALLY, and NEXT.

When a formula occurs as one of the lines of a proof, that occurrence is called a *top-level occurrence*. If an occurrence is not at the top level, it has a *parent occurrence*, which is the smallest occurrence containing the given occurrence. For example, consider the formula $w \supset \Diamond(x \vee y)$. The parent occurrence of x is $x \vee y$. The parent of $x \vee y$ in turn is $\Diamond(x \vee y)$, and the parent of that formula is $w \supset \Diamond(x \vee y)$.

I use the term *parent formula* when the content of the parent occurrence, rather than its position, is being considered.

One point needs clarification. I always consider a conjunction or disjunction of more than two terms to be a single formula, rather than an abbreviation for a more complex, parenthesized formula. This means the parent formula of the occurrence of y in $x \wedge y \wedge z$ is $x \wedge y \wedge z$ (not $y \wedge z$, for example).

The occurrence data structure is intended to list the formula, its parent formula, the parent's parent, and so on until the top-level is reached. We could propose the recursive definition:

> An occurrence of a formula is represented by a pair whose
> car represents the formula itself, and
> cdr is nil if the occurrence is top-level, or
> represents the parent occurrence otherwise.

However this definition is not good enough. It cannot distinguish between the two occurrences of A in $A \wedge B \wedge A$, for example. To remedy this flaw, suppose we are given a formula, and an

occurrence of one of its arguments. Define the *tail* of the formula beginning at that argument to be the list of arguments from the given argument onward. For example, the tail beginning at the first A in $A \wedge B \wedge A$ is (A B A), while the tail beginning at the second A is (A). When the formula is represented as a list, as described above, this "tail" will really be one of the tails of the list.

Now we are able to describe the occurrence data structure.

> A top-level occurrence is represented by a list containing a list containing the formula
>
> Other occurrences are represented by a pair whose
>
> car is the tail of the parent formula, starting at the formula
>
> cdr represents the parent occurrence

Example

Suppose the formula $(A \wedge B) \supset \neg C$ occurs as a line in the proof. This formula is represented internally as

(IMPLIES (AND A B) (NOT C)).

The occurrence of this formula is a top-level occurrence, and is

(((IMPLIES (AND A B) (NOT C)))).

The occurrence of $A \wedge B$ within the formula is represented as

(((AND A B) (NOT C)) ((IMPLIES (AND A B) (NOT C)))).

The occurrence of C is represented as

((C) ((NOT C)) ((IMPLIES (AND A B) (NOT C)))).

Sharing Storage

The occurrence structure often contains duplicated formulas, and it is natural to try to share list structure between them. To see how this structure can be shared, it is best to give a new description of occurrences.

The list structure representing a formula is actually a binary tree. Roughly speaking, an occurrence marks a point within this binary tree. More precisely, if the links in the binary tree were bidirectional, then an occurrence could be represented by a pointer to one of the cells in the tree. But there is a disadvantage to making those pointers bidirectional — it would make it impossible to share structure between two different formulas. Hence, whenever a deduction is performed, subformulas of the premises would need to be copied before they could be used in the conclusion.

An occurrence is an alternative way to implemen t the required bidirectional links. It can be seen ***as a list of pointers into the formula structure.*** The first pointer in the list is to a cell in the list structure. The second pointer is to that cell's "parent", and SO 011. One must be careful to define the "parent" correctly. (Recall that we had to revise our first definition of occurrence, because it did not give enough information. One way to characterize this error is by saying that we chose the wrong definition of "parent".) Top-level formulas and atomic formulas must be treated as special cases, because we need to create additional cells ou tside the formula itself, but iii general this view of an occurrence is correct.

The parser and prettyprinter always create occurrences using this model, so that occurrences use as little storage 8s feasible. However, the other algorithms manipulating occurrences do not rely on the sharing.

4.2 Algorit hrns

The operations needed for manipulating formulas arc simple, and will not be mentioned further here.

We will need several functions on occurrcnccs:

- Given an occurrence, return the formula of which it is an occurrence.
- Given a.11 occurrence, find the number of ◯ operators in which it is nested, and check whether it is in the scope of ◇ or ❑ . A similar operation finds the polarity of an occurrence.
- Given an occurrence and a formula, re t urn the formula that results from replacing the occurrence by the formula.

The defini tion of occurrence makes all these operations very simple.

Given an occurrence its corresponding formula is found by taking the `caar` of the occurrence. If the formula is compound, its operator and argument list arc found by taking a further car and cdr, respectively. (This is not, to say that occurrences arc manipulated using car and cdr. All these operations are actually implemented by appropriate macros.)

Repeatedly taking the cdr of an occurrence will reveal the successive parent occurrences and the parent formulas are given by taking the `caar`. Hence, the number of ◯'s in which an occurrence is nested is easily calculated by stepping through the parent formulas counting how many ◯'s are found.

The polarity of an occurrence can be found in a similar way, though there is a complication because the antecedent of an implication has negative polarity, but the consequent has positive polarity. The algorithm can determine which of these two cases holds quite simply. Suppose we are given an occurrence whose parent occurrence i s an implication. If this is an occurrence of the antecedent, then the car of the occurrence will be a list, holding both the antecedent and the consequent. 0 therwise, the car of the occurrence will be a list, holding just the consequent. Now the parent formula of an occurrence is given by the caadr of the occurrence, and the cdr of this formula will be a list containing the antecedent and consequent. So, if the cdaadr of the occurrence is the same as its car, if and only if this is *il.11* occurrence of the antecedent.

Determining whether one occurrence lies within another is simply a matter of checking whether the first occurrence is a tail of the second.

Finally consider how to replace an occurrence by a different formula. This opera tion could be defined so that it returned an *occurrence* of the new formula within the changed outermost formula. However, it is simpler to return the outermost formula itself, and this turns out to be sufficient for our purposes. (If we used destructive operations on occurrences it would be a simple matter to change the given occurrence into an occurrence of the new formula. However occurrences share storage between one another, so destructive operations have unwanted side-effects.)

s o suppose weare given an occurrence and a formula that is to replace this occurrence. If the occurrence i s already at the top level, we can return the new formula and stop. Otherwise we calculate the formula that must replace the parent occurrence, and call the replace function recursively. Wc already know the tail of the parent formula, so we can calculate the formula that

must replace it by first calculating the new tail of the parent formula, then taking the parent formula, and replacing the existing tail by the new one. (All this must be done non-destructively, but there is still some opportunity to share storage, and this is done wherever possible.)

5. Interface to Zmacs

Zmacs was used for the user interface because it made many facilities available to the user for little cost. However, the various structures and services used by the editor should be used carefully by PTL to avoid problems caused by changes to Zmacs. I decided to use existing editor structures wherever possible, rather than defining slightly different ones especially for PTL, and to use editor functions at the highest level possible.

In accord with these principles, PTL mode shares many functions with lisp mode, but, still, two Zmacs-related functions had to be specially written to support PTL. The first is the buffer sectionization algorithm, which determines the boundaries of forrnulas within the buffer. The second is the group of functions that control selection of forrnulas using the mouse.

The reader is assumed familiar with the Zmacs internals described in Appendix A of this report.

5.1 Sectionization

A PTL-mode buffer is sectionized. Standard section-nodes are used for the sections. Sections belong to one of two classes, one for formulas and one for general text. The definition-type field defines whether a node holds a formula or text. The function-spec field is used to hold the line number of a formula node.

A formula node begins at the first character on the first line of the formula, the line containing the line number. The formula ends just, *before* the carriage return in the last line of the formula. This final carriage return always begins a new text node. Thus, there is always a text node between any two formula nodes. If the next line also contains a formula then the text node contains just the carriage return between the lines. If the next line contains text, the text node includes the text lines as well.

This convention is adopted because the only way that I'TL knows that the region list for a formula node is valid, is by checking the node-tick against the compile-tick for the node. If two formula nodes are adjacent, and the user inserts a new line between them, the node-tick of the first node will be updated, even though the region list, is still valid. To avoid unnecessary reparsing, a formula node is defined to end just before the carriage return. If a new line is inserted, the text node containing the carriage return is updated, not the formula node before it.

(In fact an earlier version of this program made formula nodes read-only to prevent unintended changes. This was abandoned for two reasons. It, was sometimes inconvenient to use, and it activated a bug in Zmacs which crashed the editor if one attempted to delete a region containing one of the read-only nodes.)

5.2 Using the Mouse to Select Formulas

When the mouse is being used to select formulas, three operations must be carried out. First, the physical position of the mouse cursor on the screen must be discovered. Second, this physical position is used to determine the occurrence that is being pointed to. Thirdly, an outline is drawn around the formula.

Finding the Mouse Position

The mouse cursor position is found using the Zmacs internal function `mouse-char` as described in Appendix A.

Calculating the Formula Being Referenced

To relate the physical position to a formula, each formula section has a property named :msregion-list. The value associated with this property is a list of mouse-sensitive regions, arranged so that the smaller regions always appear before their parent region. This property is initialized when the formula is parsed, or when a new formula is printed. Every deduction command begins by checking that these properties are current, calling the parser if necessary to update them.

Each mouse-sensitive region is defined by a st `ruct` called msregion, which contains two BP's marking the beginning and end of the region and a value, which is the occurrence (see above) that this region represents.

A single formula can occupy several lines of the file. We cannot assume that the surrounding region's boundaries will be on the same line as the mouse cursor, so determining whether the mouse position lies within any region may involve a search through the lines of the buffer. However, Zmacs provides an internal function, `BP-<` for just this type of situation. It is passed two BP's, and returns T if the first BP is earlier in the text than the second, regardless of whether the two BP's are on the same line or not.

So to find the formula being pointed to, first create a temporary BP at the current mouse position. Then search through the region list of the formula node containing that BP, for the first region surrounding the BP. This region will be the required occurrence.

Outlining Formulas

Single forinulas spanning multiple lines of text also affect the outlining of forinulas on the screen. To outline a multiple line formula, a hollow rectangular blinker is created for each line of the formula. Blinkers are XORed onto the screen, and the bottom bar of such a blinker coincides with the top bar of one 011 the next line. Therefore, when several blinkers are placed on the screen, their common boundaries cancel each o ther out, and only the ou tline remains.

The blinkers are kept on a list, and re-used rather than being re-created. Normally, all blinkers on the list are made invisible before beginning a new outline. This is wlly the outline appears to move from one place to another. For *persistent* blinkers, which are the blinkers outlining the first of two premises, a separate list is used so that they remain visible un til expli tly switched off.

6. Pretty Printing

Since derived formulas may be too long to be printed on one line, the printing routines must be able to do more than just translate a formula in internal form into its external form. They must be able to decide appropriate line breaks.

On the other hand, printing formulas is not so complex that specialized algorithms are needed for them. So a printing algorithm of medium complexity is required.

The prettyprinter actually used is based on the printer presented by Oppen in [2]. It uses limited space, and requires few enhancements to print formulas adequately.

6.1 Structured Text

The prettyprinter is passed some structure defined on the text to be printed, as well as the text itself. The text is considered *as* a stream of *units,* where a unit is defined recursively as either

- A string, or
- A sequence, $u_1\, u_2 \ldots u_n$, where each u_i is either a unit or a ***space.*** (Spaces are optional line breaks.)

I disallow consecutive spaces and spaces at the beginning or end of a unit.

This structured text is passed to to the prettyprinter as a stream of tokens, where the tokens can be

- Strings,
- Spaces,
- Brackets, written [and], which group the strings and spaces into logical units.

If a logical unit of text cannot be printed on a single line, it will be split into Several lines. Each continuation line will be indented two additional characters from the indentation of the first line of the logical unit.

If a space token is not converted into a line break, it is printed as a single space.

6.2 Printing Algorithm

Printing can be considered as being divided between two processes, called *scan* and *print.* (Functions called scan and print are not actually implemented. This is only a description of the algorithm being used.)

The input stream is fed into *scan,* which calculates a size for each token in the input stream, and then passes the token and its corresponding size on to *print.* I measure sizes as numbers of characters here, though the generalization to Variable-Width characters is easy.

Sizes of each Token

The size of a string is the number of characters in the string. The size of a] is zero.

The size of a [is the space required to print everything in the logical unit, and everything in units following it up to the next space token. Assume that no spaces nested within the logical units are converted to line breaks. For example, suppose $u_1\, u_2 u_{3\sqcup}$ occurs in the text, where $\sqcup$

is a space, and each unit u_i contains n_i characters. The size of the [that begins the u_1 unit will be $n_1 + n_2 + n_3$. For u_2 the corresponding size will be $n_2 + n_3$ and for u_3 the size will be n_3.

The size of a space is 1 greater than the size of the string or [that follows it. Hence, the size of a space is the number of characters that would be needed to print the space, and everything up to the next space, assuming that no spaces at any lower level of nesting are converted into line breaks.

How Print Processes the Text

Print takes the stream of text to be printed, together with the associated sizes and prints the text as follows. When a string token is rcccivcd, the string is simply printed. When a [is received the current print position is recorded on a push-down stack, and any new line started will be indented to the value at the top of the stack, plus 2 spaces. If a] is received this stack is popped.

When a space token is received the printer must decide whether to convert the space into a line break. It does this by examining the size associated with the space. If enough room remains in the current line to accommodate this length, the space is printed as a space. Otherwise a new line is begun, indented as described above.

Improving Scan's Performance

Scan calculates the size for each item by holding all items in a FIFO buffer. The first item in the buffer is passed to *print* as soon as its size is known. Hence strings and]'s can be passed as soon as they reach the head of the buffer. ['s and spaces must be held in the buffer until a space at a high enough level of nesting is received.

But how is *scan* to calculate the sizes of each item without needing to buffer a great deal of text? In the worst, case, where the text forms a single logical block, it appears that *scan* will need to store all the text before it can be passed to *print*.

But since an item that has unknown length is either a [or a space, *print* will only use the size to check whether the item will fit on the current line. Hence *print* only needs to know that the size is larger than the available space. The need to buffer the [indefinitely could be avoided if *scan* were permitted to read *print*'s variable recording the amount of space remaining iii the current line. *Scan* could then keep track of the minimum space needed to print the contents of the buffer. If this minimum space exceeded the available space, the first item in the buffer could be passed to the printer immediately, with a very large size.

Oppen claims ([3]) that with this provision *scan* will never require more than 3 times the line width in buffer space. However, he is clearly mistaken. Given any fixed amount of buffer space, an input stream of the form

$$[[\cdots[S_\sqcup S]_\sqcup S]\cdots{}_\sqcup S]$$

can be constructed to use up all the buffer space just to store the initial sequence of ['s. This example cannot be dismissed as one which could not be printed at all. It prints to the same format as $[S_\sqcup S_\sqcup \ldots S]$.

[Oppen's claim that this algorithm uses $O($ *width*$)$ space can, however, be rescued by performing some extra processing. One possibili ty is to encode consecutive ['s and consecutive]'s so that an unbroken sequence of bracke ts occupies only one place in the buffer.]

6.3 Implementation

The printer actually implemented includes a number of extensions to this basic algorithm:

- provision for printing comments as well as the formatted text,
- building a list of the mouse-sensitive regions within the text. (each unit bounded by [and] will become mouse sensitive.) This provides the :msregion-list required by the Zmacs interface, as described in chapter 5.
- provision to use formatted output in several different ways by changing the definition of *print*. (One such function prints to a Zmacs buffer, another to a string variable, and others could be defined if necessary.)

7. Parsing

The requirements for a parser are similar to the requirements for a printing algorithm. A straightforward parser that does not use resources unnecessarily is needed. Since this is a simple parsing problem, any standard parsing algorithm could be used.

SLR parsing was chosen. The parse table was hand constructed rather than generated using a parser gcncrator, and contains 25 states.

The output generated by the parser is a list of the mouse-sensitive regions within the formula. There is one mouse-sensitive region for each of the subformulas of the formula. The occurrence that each region represents is recorded with the region, and is available if the region is selected. The boundaries of the region must also be rccordcd.

7.1 Grammar

(The symbol "VAR" represents any propositional variable.)

1. F → VAR
2. F → (F)
3. F → CONJ
4. F → DISJ
5. F → OF
6. F → OF
7. F → ◇F
8. F → ¬F
9. F → F ⊃ F
10. CONJ → F ∧ F
11. CONJ → CONJ ∧ F
12. DISJ → F ∨ F
13. DISJ → DISJ ∨ F

The symbol → can be used instead of ⊃.

7.2 Parsing Algorithm Details

Although the list of mouse-sensitive regions will finally be required with inner subformulas occurring before outer formulas, it is more convenient to build the list in the opposite order, and reverse it when this is complete

The parser stack is maintained as a list of *frames*, each of which contains a start and end BP for the unit represented by the frame, and some information about its contents. Each frame also records the state of the parser at the time the frame was pushed onto the stack. This state is used in calculating the new s tate after a reduce action.

Many parser states have identical actions for all inputs. This fact is used to encode the parse table iii a special way to reduce its size.

A similar abbreviation is used to encode the state transition table, which calculates the correct parser state after a reduction has taken place. Here, rather than having many states with identical new states, each state has a valid next state for CON.1 and DISJ if and only if it has a valid next

state for F. Further, the state is mapped to state 4 if the next frame contains a CONJ, and to state 5 if the next frame contains a DISJ. In this case only the next states for a frame containing F are recorded in the state table, and the values 4, or 5 are substituted by the access function, if the frame contains a CONJ or DISJ.

For debugging, the variable trace-parser is provided. Setting it to a non-nil value will provide a trace of the shift and reduce actions taken by the parser.

Acknowledgement

I wish to thank Martin Abadi for suggesting this project, and for his many comments and suggestions.

References

[1] Martin Abadi and Zohar Manna. "Nonclausal Temporal Deduction" in *Proc. Conf.* on *Logics of* Programs, Brooklyn, June 1985. Springer Verlag Lcct. Notes in Comp. Sci. 193. Ed. Rohit Parikh. Also available as Report No. STAN-CS-85-956, Computer Science Department, Stanford University.

[2] Alfred V. Aho and Jeffrey D. Ullman, Principles **of Compiler** Design, Addison-Wesley, 1979.

[3] Derek C. Oppen, **Pretty Printing,** Report No. STAN-CS-79-770, Computer Science Department, Stanford University, Oct. 1979.

Appendix A: Overview of Zmacs Internals

Here, I describe internals of Zwei and Zmacs that arc not currently documented. Further information can be obtained from appropriate sections of the source code for Zmacs. The reader who wishes to browse Zmacs source code is advised to learn the Zmacs commands Edit Zmacs, and Edit Definition.

I assume that the reader is reasonably familiar with Zmacs and has a good knowledge of the array handling and flavor system of Zetalisp.

All functions and structures mentioned arc contained in the **zwei** package.

A.1 Data Structure for Storing Text

All text is stored as a set of strings while it is being edited. An entire file is read into virtual storage when it is opened. Each line of the file is stored as a single (variable length) string. The lines are chained together through pointers kept in the array leader of the lines.

A position within the text can bc marked by creating a structure called a BP. All BP's contain a pointer to the line, and the index of a character within that line. The "point" (at which the cursor appears) is represented by such a BP, but it is not the only BP that exists. In fact, many BP's are created during an editing session. Many functions create temporary BP's to mark a place in the text while the function is operating. Other functions create permanent BP's, which are intended to remain valid even after their creating function has completed execution. Zmacs updates permanent BP's automatically whenever text is inserted or deleted. If text is inserted in a line before a permanent BP, it will bc moved so that it still points to the same character. Text inserted after the BP has no effect on the BP. A third field determines how a BP is to be updated if text is inserted at the BP itself. A normal BP remains at the same index; a moves BP is moved by inserting text.

Often text has a logical structure that is independent of its division into lines. Zmacs supports operations on a logical chunk of text through a structure called a *node*. Each node contains a contiguous piece of text, bounded by two BP's recorded in the node structure. A node also contains a time-stamp recording the last update of the node (called node-tick), and may have other properties. Nodes may have subnodes up to any depth.

Nodes arc implemented as instances of a flavor. (The methods of this flavor are only the standard ones for retrieving and setting instance variables.) Different kinds of nodes arc built using the node flavor as a base. Such an extended node that is visible to the user is the *buffer*, which contains additional instance variables to store the file name, current editing mode and position, etc.

Another, less visible, type of node is the *section*. A typical use of section nodes is to hold the lisp function definitions in a file. A buffer that is in LISP mode is partitioned into sections, each section containing one of the lisp definitions. The additional properties of a section include the time-stamp of last compilation or evaluation of this section, the type of definition (function, *macro*, flavor, etc.) and the name of the object defined.

A.2 Deflning and Installing Commands

Commands arc defined using the def corn macro, which takes four arguments: the command name, documentation string, options list, and command body.

The command name is best explained by examples. The command known to the Zmacs user as "Save File" is defined in a def corn as COM-SAVE-FILE, "Compile Buffer" is defined as COM-COMPILE-BUFFER, "Simplify Formula" is defined as COM-SIMPLIFY-FORMULA, and so forth.

The documentation string is the text that will be displayed if the user issues the HELP C command to get a description of the command.

The options list controls the effect issuing the command will have on a region if oue is current. All PTL commands take the default, which is to de-activate the region.

The body is a series of lisp forms to be executed when the command is called. The value of the final form is used as a hint to the screen update routines. dis-none, for example should be returned if the command does not change any text. dis-text means that the command has changed the text displayed.

A command can be used as an extended command (via Meta-X) as soon as it has been defined with def corn. It can be bound to a particular key using the "Install Command" command, or it can automatically be associated with an editor mode, by having it placed in the mode's command table.

A.3 Editor Modes

Each editor mode, such as Lisp mode or text mode, is defined by a flavor. This flavor will only be instantiated once, at the time that the first buffer in that mode is created.

There are some 20 to 30 methods that the mode must define, but many of them may be given defaults by including the base flavor major-mode. These methods define such things as the format of comments in the edited text, how matching parentheses are defined, how the buffer is to be sectionized, and code to be executed when a buffer of the mode is edited or left. A special command table, *mode-comtab*, may bc set up in this initialization code. This will set up key bindings that are valid in any buffer of that mode.

The PTL-mode flavor is based on a copy of Lisp Mode.

A.4 Mouse Interface

The position of the mouse cursor can be determined at any time by the function mouse-char. This must be passed a pointer to the editor window structure, and returns five values: the character to which the mouse is pointing, its x and y co-ordinates (in yixcls), the line containing the mouse, and the index of the character within the line. The window is available as the global variable *window*, or as an argument passed to a blinker handler, as described in the next paragraph.

Normally Zmacs simply draws a box around the character to which the mouse points, This action can be changed by re-binding the variable *global-mouse-char-blinkor-handler* The function bound to this variable is called each time the mouse is moved, and it normally is used to set blinkers on the screen. The function will be called with seven arguments: a blinker to use, the current window, and the 5 values listed above to specify the current mouse position. The blinker handler will be called from within the mouse process, not, from within Zmacs, so it, does not have access to local Zmacs variables. However, it does have access to global variables, and can follow pointers from the line structure that it has been passed.

A number of similar global variables control the documentation string that is displayed at the bottom Of the screen, and the character to use for the mouse pointer.