

# Computing PLA, version 1: Description of the algorithms and interface

Christopher Potts

May 16, 2006

## Contents

<b>1</b>	<b>Background</b>	<b>2</b>
1.1	Broad overview of this document . . . . .	2
1.2	Why I wrote Computing PLA . . . . .	2
1.3	A few notational conventions . . . . .	3
<b>2</b>	<b>Set up</b>	<b>3</b>
2.1	User input . . . . .	3
2.2	The central data structure . . . . .	4
2.3	Predicate denotations . . . . .	4
2.4	Well-formed formulae . . . . .	5
2.5	STATE EXTENSION . . . . .	5
<b>3</b>	<b>Syntax checking</b>	<b>6</b>
<b>4</b>	<b>Interpretation</b>	<b>7</b>
4.1	INTERPRET . . . . .	7
4.2	NEGATION . . . . .	8
4.3	EXISTENTIAL . . . . .	9
4.4	CONJUNCTION . . . . .	10
4.5	ATOMIC FORMULAE . . . . .	10
<b>5</b>	<b>CGI Design</b>	<b>11</b>
5.1	Start: HTML form . . . . .	11
5.2	Initial output: pla.cgi . . . . .	11
5.3	Continued output: pla-continue.cgi . . . . .	12
5.4	Summary picture . . . . .	12
<b>6</b>	<b>Limitations not shared by Dekker's logic</b>	<b>12</b>
<b>7</b>	<b>Equivalences and useful inputs to try</b>	<b>13</b>

# 1 Background

## 1.1 Broad overview of this document

This document describes the algorithms involved in `Computing PLA`, a computational implementation of Dekker 1994. It also describes some minor limitations of the program not shared by Dekker's logic. It is meant to be read after study of Dekker 1994 and perhaps near the start of one's time hacking around with `Computing PLA`.

### The CGI

<http://people.umass.edu/potts/computation/>

The program is written in Perl, with Web support from the CGI module.

## 1.2 Why I wrote `Computing PLA`

It is mainly a pedagogical tool.

Dekker's PLA is wonderful for teaching dynamic semantics and exploring its theoretical virtues. The core system is classical, so we keep a firm grip on what we know from the static realm, but existentials and pronouns are dynamic. Their denotations change as the information state changes, and this is as it should be.

However, it's somewhat hard to teach the paper effectively. One ends up spending a lot of time drawing states on the board and then erasing or modifying them as new formulae arrive. It's hard on students' notebooks, and it slows the discussion to a crawl.

With this script, though, I can very quickly and easily update the information state with new formulae. I can work through important equivalences. I can take requests from the audience. My students can go off and try their own update sequences. And so forth. To my mind, it's a prime example of the fruitfulness of doing computation for linguistics.

### 1.3 A few notational conventions

- $i \leftarrow j$  means that  $i$  is assigned  $j$ 's value.
- Comments in algorithm descriptions are set off with  $\triangleright$ .
- $S_a$  is an information state, informally labeled  $a$ ; the corresponding data structure is an array of arrays.
- $S_a[i]$  is the  $i$ th position in the array  $S_a$  (with counting beginning at 0).
- $S_a[i][j]$  is the  $j$ th position in  $i$ th array in  $S_a$  (with counting beginning at 0).
- $\varphi$  and  $\psi$  are well-formed formulae.
- $d$  is a domain size.
- Subroutine names are printed in small caps.

## 2 Set up

### 2.1 User input

The user supplies three inputs at the start.

**Domain size: A natural number** The elements in the domain are natural numbers. When the user specifies a domain size  $n$ , she is actually specifying a domain consisting of the natural numbers  $0 \dots n - 1$ .

The script manipulates natural numbers as its individuals purely for the sake of convenience. The most important advantage of talking about numbers is that we can define the predicates of the language as predicates of natural numbers on their usual interpretation: `even`, `prime` and the like. (See section 2.3.)

**Sequence length: A natural number** The user inputs this number only for convenience. It sets up the information state. If she enters 1, then we begin with a set of single-membered sequences — one for each entity in the domain. If she enters 2, we begin with the set of all pairs of entities, i.e.,  $D^2$ . And so forth. See section 2.5.

The choice here has few consequences because the user can use existential statements of the form `(Ex : (number x))` to uniformly extend the domain (since `number` always denotes the entire domain).

If the domain of individuals is  $0 \dots 5$  and the sequence length is 6, then the script must generate  $6^6 = 46,656$  sequences. Permutation algorithms are slow, though the present script is about as good as it gets in terms of its algorithm (section 2.5).

To avoid slowdowns and server problems, the script checks to ensure that, where  $n$  is the domain size and  $k$  is the sequence length, the value of  $k^n$  is below 500 in the initial state and remains below 1000 for all later states. The algorithms are prepared for any size inputs, though, and users wishing to work with very large states might contact me about obtaining a downloadable version with a command-line interface.

**Formulae** At the start, along with the above two background parameters, the user supplies a formula. (See section 2.4 for more on the set of well-formed formulae.)

In later updates, the user enters just new formulae. The domain size is fixed throughout the dynamic interpretation run, and the sequence length is, at every point after the start, determined by the number of existentials that the user enters: if the initial sequence length is  $n$  and the user has entered  $k$  existentials, then the total sequence length at that point is  $n + k$ .

## 2.2 The central data structure

Information states are arrays of arrays of numbers. Each time an update is performed, this state is copied. The copy is modified according to the user's formula, and then both the input and the output states are displayed in an HTML page, along with a window asking for another formula.

In Dekker's logic, sequences (arrays) are eliminated by certain updates. For pedagogical purposes, no array is ever eliminated by the script. Rather, arrays are overwritten with dashes, one for each entity in the sequence. These act as placeholders for eliminated states, making it easier to see how a formula acted on the input state.

## 2.3 Predicate denotations

The lexicon is a function that maps predicates to functions from entities in the domain into  $\{\mathbb{T}, \mathbb{F}\}$ .

- $n$  is a number iff  $n$  is in the domain.
- $n$  is null iff  $n = 0$ .
- $n$  is even if  $n = 0 \bmod 2$ , else  $n$  is odd.
- $n$  is prime if it is 2 or there is an integer  $i$ ,  $1 < i < n$ , such that  $n = 0 \bmod i$ , else  $n$  is composite

## 2.4 Well-formed formulae

The script does not yet support abbreviations of the connectives, but the equivalences of PLA, which are classical, obtain also in the script:

LEXICON:

```
    preds: number, zero, even, odd, prime, composite
    args: 0 ... n-1 [n = user supplied domain size]
    pronouns: p0 ... k-1 [k = current sequence length]
    variables: x,y,z (must be bound)
```

WELL-FORMED FORMULAE (WFFs)

where  $X$  and  $Y$  are args or pronouns,  $B$  and  $C$  are WFFs, and  $x$  is a variable

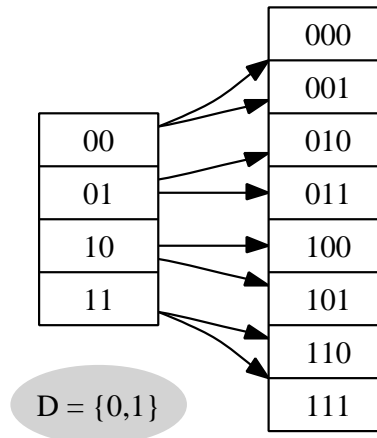
(pred $X$ )	[atomic]	universals:	$\sim\text{Ex}:\sim B$
$\sim B$	[negation]	implications:	$\sim(B \ \& \ \sim C)$
$(X = Y)$	[equality]	disjunctions:	$\sim(\sim B \ \& \ \sim C)$
$(B \ \& \ C)$	[conjunction]		
$(\text{Ex}:B)$	[existential]		

## 2.5 STATE EXTENSION

The script defines the initial state to be the full set of one-membered sequences of elements in the domain. It then extends those sequences  $k - 1$  times, where  $k$  is the sequence length input by the user. The extension algorithm, used also for the existential, is defined as follows ( $d$  is a domain size):

```
STATE EXTENSION( $S_{\text{input}}, d$ )
(1) 1 initialize  $S_{\text{output}}$ 
    2 for  $s \in S_{\text{input}}$ 
    3   do for  $j \in 0..d-1$ 
    4     do  $s' \leftarrow s$   $\triangleright$  (re)set  $s'$  to the original sequence
    5     push  $j$  onto  $s'$ 
    6     push  $s'$  onto the output state  $S_{\text{output}}$ 
    7 return  $S_{\text{output}}$ 
```

Here is a picture of the way a single operation of STATE EXTENSION works on a state that is the permutation closure of  $\{0, 1\}$ :



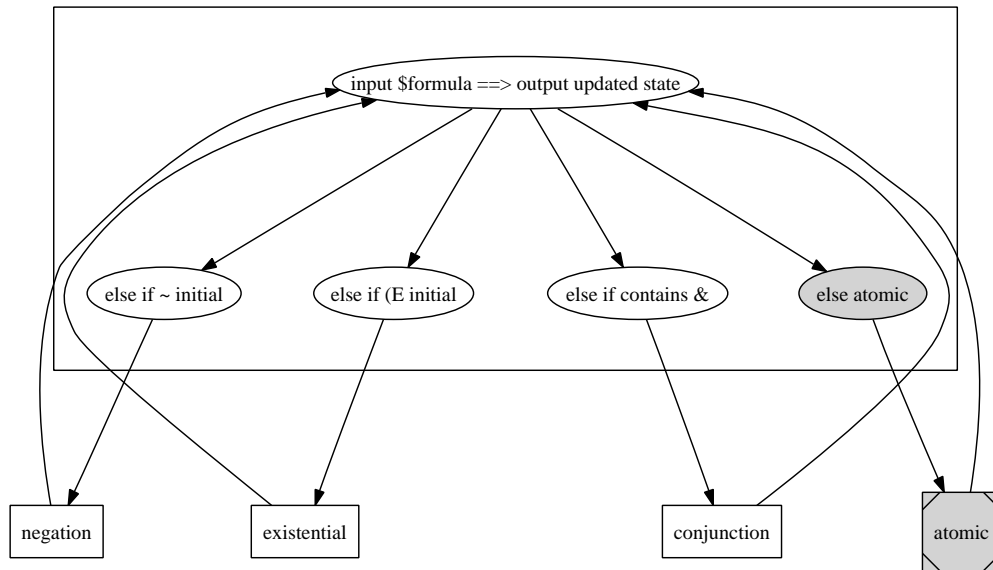
### 3 Syntax checking

Each time the user enters a new formula, it is subjected to a parse-check procedure. This is the purvue of the subroutine `PARSE CHECK` and it associated subroutines. I won't take the time to describe the algorithms for this here. They are not very interesting, and they let some bad stuff slip by. (As far as I know, they block no grammatical stuff.)

It's among my short term plans to improve this aspect of the script by using a general checking mechanism like the Earley parser. For now, if your domain is under 6 and things aren't happening very fast, then either your Net connection is out or the script is endlessly looping on your agrammatical input.

## 4 Interpretation

This is the intellectual and computational meat of the scripts. The general picture is as follows: a formula that passes the PARSE CHECK algorithm is fed to a master subroutine INTERPRET, which calls various subroutines, as appropriate (subroutines are boxed):



### 4.1 INTERPRET

The main subroutine is INTERPRET. Its inputs are a state  $S_{\text{input}}$  and a formula  $\varphi$ .

- ```
INTERPRET( $S_{\text{input}}, \varphi$ )
(2) 1 copy  $S_{\text{input}}$  to  $S_{\text{output}}$ 
    2 if  $\varphi$  begins with  $\sim$ 
    3   then  $S_{\text{output}} = \text{NEGATION}(S_{\text{output}}, \varphi)$ 
    4 elseif  $\varphi$  begins with ( E
    5   then  $S_{\text{output}} = \text{EXISTENTIAL}(S_{\text{output}}, \varphi)$ 
    6 elseif  $\varphi$  contains &
    7   then  $S_{\text{output}} = \text{CONJUNCTION}(S_{\text{output}}, \varphi)$ 
    8 else  $S_{\text{output}} = \text{ATOMIC}(S_{\text{output}}, \varphi)$ 
    9 return  $S_{\text{output}}$ 
```

## 4.2 NEGATION

A formula sent here has its negation stripped off. The result is sent back to INTERPRET. That interpretation is stored in an auxiliary state  $S_{\text{positive}}$ . This is compared with the input state: if a sequence  $s$  is dashed out (removed) in  $S_{\text{positive}}$ , then  $s$  is included in  $S_{\text{output}}$ , else  $s$  is dashed out:

```

NEGATION( $S_{\text{input}}, \sim \varphi$ )
(3) 1  copy  $S_{\text{input}}$  to  $S_{\text{output}}$ 
     2   $S_{\text{positive}} = \text{INTERPRET}(S_{\text{output}}, \varphi)$       ▷ interpret  $\varphi$  with  $\sim$  removed
     3  for  $i \in 0..length(S_{\text{positive}})$ 
     4      do if  $S_{\text{positive}}[i]$  contains only dashes
     5          then  $v \leftarrow 1$ 
     6          else  $v \leftarrow 0$ 
           ▷ for existentials
     7          if  $\varphi$  is an existential
     8          then if  $v = 0$ 
     9              then for  $i \in 0..length(S_{\text{output}})$ 
    10                  do for  $j \in 0..length(S_{\text{output}}[i])$ 
    11                      do  $S_{\text{output}}[i][j] \leftarrow " - "$ 
    12                  return  $S_{\text{output}}$ 
           ▷ for all other formulae
    13          elseif  $v = 0$ 
    14              then replace every element in  $S_{\text{output}}[i]$  with " - "
    15  return  $S_{\text{output}}$ 

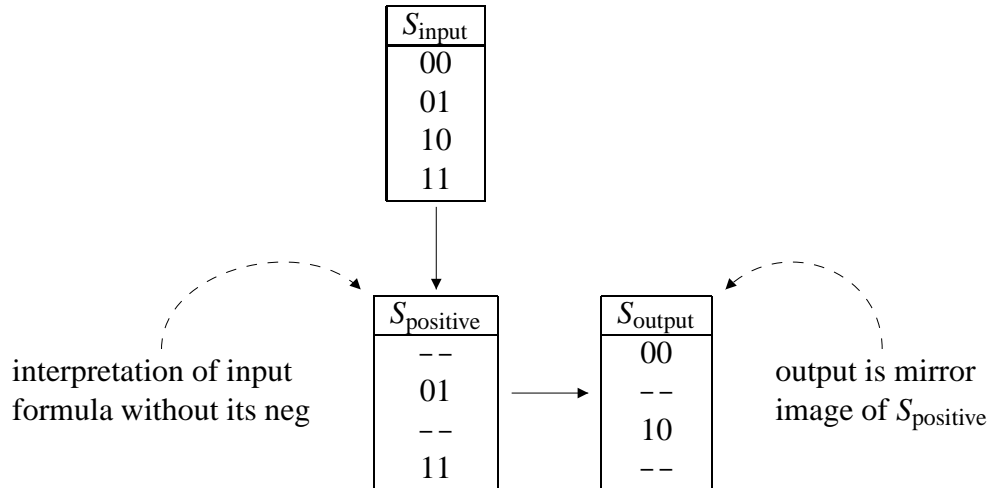
```

The subroutine handles existentials differently from regular atomic formula.

**For existentials, see lines 7–12** For existentials, negation is a pure test: if  $\sim \text{Ex} : B$  is true, then there is no element in the domain with property B, and if some element in the domain is a B, then the formula is false.

So at line 4, we begin moving through the positive state. We check to see if the current sequence is dashed-out. If it isn't (line 8), then something has the property named by the scope formula, so we wipe out the entire output state (lines 8–11), and we return that totally wiped out state.

**For all other formulae, see lines 13–14** At this point, we are still moving through the sequences in  $S_{\text{positive}}$ . If a state is dashed out there, then it is left alone in the output state. If a state survives undashed into  $S_{\text{positive}}$ , then it is wiped out in  $S_{\text{output}}$ .



### 4.3 EXISTENTIAL

Existentials have two components in PLA:

1. The quantifier itself uniformly extends every sequence in the current state in every possible way, as described in section 2.5
2. The new information state is updated with the scope property, with occurrences of the quantifier's variable in that scope replaced by appropriately indexed pronouns.

We depend on a globally-set variable  $ec$ . It is reset to 0 at the start of every formula update. After an existential is updated, it is incremented by 1. So if a formula contains three existentials, then  $ec = 2$  at the end of the evaluation, at which point it is returned to 0.

EXISTENTIAL( $S_{\text{input}}, \text{Ex} : \varphi$ )

- (4) 1 **copy**  $S_{\text{input}}$  to  $S_{\text{output}}$
- 2  $S_{\text{output}} = \text{STATE EXTENSION}(S_{\text{output}}, d)$
- 3 **split**  $\text{Ex} : \varphi$  into  $\text{Ex}$  and  $\varphi$
- 4 **replace** every occurrence of  $x$  is  $\varphi$  with  $p_{ec}$
- 5  $ec = ec + 1$
- 6  $S_{\text{output}} = \text{INTERPRET}(S_{\text{output}}, \varphi)$
- 7 **return**  $S_{\text{output}}$

## 4.4 CONJUNCTION

In general, the CONJUNCTION subroutine finds the main connective, uses that information to split the formula into its two conjuncts, and then sends those conjuncts back to INTERPRET. The conjuncts are processed sequentially, and thus the operation is not commutative (see section 7).

In the script, the business of finding the main connective is the purview of a separate subroutine, MAIN CONNECTIVE. But it is conceptually part of the CONJUNCTION subroutine, so I include it here:

```
CONJUNCTION( $S_{\text{input}}, \varphi$ )
(5) 1 copy  $S_{\text{input}}$  to  $S_{\text{output}}$ 
       $\triangleright$  Find the main connective
     2  $b \leftarrow 0$   $\triangleright$  Set the bracket count to 0
     3 for  $i \in 0 \dots \text{length}(\varphi)$ 
     4     do if  $\varphi[i]$  is a left bracket
     5         then  $b = b + 1$ 
     6         elseif  $\varphi[i]$  is a right bracket
     7             then  $b = b - 1$ 
     8         if  $b = 1$  and  $i > 2$ 
     9             then  $m = i + 2$   $\triangleright$   $i$  is  $)$ ;  $i + 1$  is a space;  $m$  is the connective
       $\triangleright$  Get the conjuncts and interpret them
    10 split  $\varphi$  on position  $m$  into subformulae  $l$  and  $r$ 
    11  $S_{\text{output}} = \text{INTERPRET}(S_{\text{output}}, l)$ 
    12  $S_{\text{output}} = \text{INTERPRET}(S_{\text{output}}, r)$ 
    13 return  $S_{\text{output}}$ 
```

## 4.5 ATOMIC FORMULAE

There are two kinds of atomic formula: those that apply a predicate to an argument, and those that assert the equality of two terms (arguments, pronouns, or mixtures thereof). The subroutine ATOMIC FORMULAE handles them both, though, for convenience, equalities are shipped off to their own subsubroutine, EQUALITY.

Both predicate–argument and equality atoms work in basically the same way, so I review here only the predicate–argument evaluation algorithm.

```

    ATOMIC FORMULAE( $S_{\text{input}}, \varphi$ )
(6)  1  copy  $S_{\text{input}}$  to  $S_{\text{output}}$ 
     2  split  $\varphi$  into its predicate  $P$  and its argument  $\text{arg}$ 
     3  for  $s \in 0..length(S_{\text{output}})$ 
     4      do if  $\text{arg}$  is a pronoun  $pn$             $\triangleright$  for pronominal arguments
     5          then get the index  $n$ 
     6              if  $\text{lexicon}(P)(S_{\text{output}}[s][l-n]) = F$ 
     7                  then for  $j \in 0..length(S_{\text{output}}[s])$ 
     8                      do  $S_{\text{output}}[s][j] \leftarrow " - "$ 
     9          elseif  $\text{lexicon}(P)(\text{arg}) = F$   $\triangleright$  for constants
    10              do for  $j \in 0..length S_{\text{output}}[s]$ 
    11                   $S_{\text{output}}[s][j] \leftarrow " - "$ 
    12  return  $S_{\text{output}}$ 

```

## 5 CGI Design

This section provides a broad overview of the CGI aspects of this program. For a detailed understanding of how it works, it is best to look at the code itself, which I can provide.

### 5.1 Start: HTML form

The entrance to the program is an HTML page containing a form (in addition to some words of explanation about how the logic works). The user supplies the three inputs described in section 2.1. These are passed to the Perl script `pla.cgi`, where they become Perl variables with the help of the CGI module.

### 5.2 Initial output: `pla.cgi`

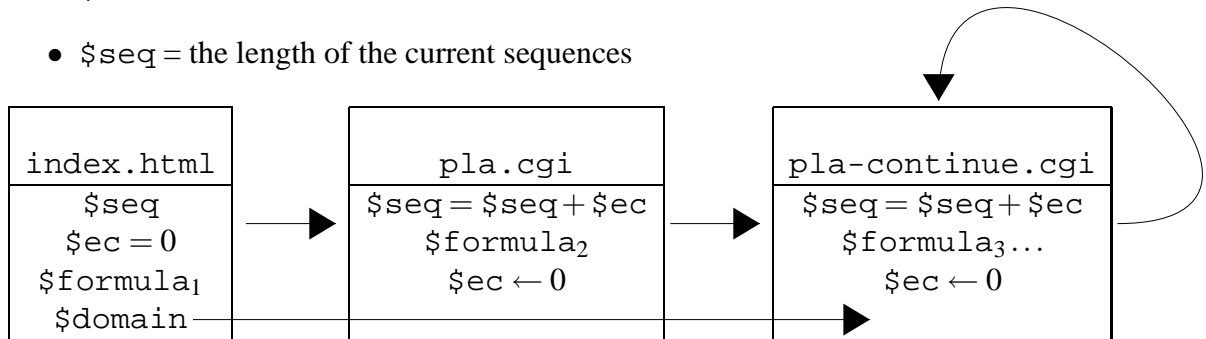
This script processes the user's initial input and outputs an HTML page that displays the input state, the user's formula, and the result of its update. There is also another HTML form. This one asks only for a formula. But the form contains hidden fields that store the values for the domain size and the sequence length as they were when `pla.cgi` finished. The form sends these variables to `pla-continue.cgi`, which outputs an HTML page that has the same format as the output of `pla.cgi`.

### 5.3 Continued output: `pla-continue.cgi`

`pla-continue.cgi` has output that is basically the same as that of `pla.cgi`. However, its HTML form sends its variables back to itself. So we are in a loop. The user can continue updating new formula until she gets tired, the state gets too big for the server to handle, or the sun grows cold.

### 5.4 Summary picture

- $\$sec$  = the number of existentials in the current formula
- $\$seq$  = the length of the current sequences



## 6 Limitations not shared by Dekker's logic

- Computing PLA has no unbound variables and no variable assignments. The only variables occur in existentials, and these are mapped to pronouns for the purposes of interpretation.
- The present version has no support for direct input of disjunctions, universals, or conditionals. This will change in the next version, though.
- The present version has no support for  $n$ -place relations for  $n$  greater than 1 (except for equality). This is a minor limitation, and if it proves too constraining, it could be changed with a bit of work in the area of ATOMIC FORMULAE. (It would be nice to inspect reasonable donkey sentences; see section 7 for the best one can do now.)

I addressed limitations due to server load and the like in section 2.1. The algorithms are in principle ready for any finite input, just as in Dekker's logic.

## 7 Equivalences and useful inputs to try

```
# DPL-style scope extension
((Ex:(even x)) & (p0 = 2))
    iff
(Ex:((even x) & (x = 2)))

# noncommutative conjunction
((p0 = 1) & (Ex:(even x)))
    is different from
((Ex:(even x)) & (p0 = 1))

# donkey sentence
# 'if every number equals a number, then it equals it'
~((Ex:(Ey:(x = y))) & ~(p0 = p1))

# nested existentials
(Ex:(Ey:(Ez:((prime x) & ((odd y) & (null z))))))

(Ex:(Ey:(Ez:((x = 0) & ((y = 1) & (z = 2))))))
```

## References

Dekker, Paul. 1994. Predicate logic with anaphora. In Lynn Santelmann and Mandy Harvey, eds., *Proceedings from SALT IX*, 79–95. Ithaca, NY: DMLL Publications, Cornell University.