# How to Pretty-Print a Really Long Formula

Lee Pike
leepike@galois.com

Galois Inc.

September 9, 2008

# Introduction

This is a talk about syntax.

So let me begin with a couple of less controversial points. . .

# Outline

- Who you should vote for in the next election.
- The one true religion.
- *How to pretty-print a really long formula*.

## Outline

- Who you should vote for in the next election.
- The one true religion.
- *How to pretty-print a really long formula*.

Just kidding.

# Credits

This work is completely inspired by Leslie Lamport's, *How to write a long formula*[1] (and also his *How to write a proof*[2]).

All the good ideas are Lamport's; the pedantic ones are mine. Here are our modest contributions:

- An implemented pretty-printer.
- Small simplifications.
- Formatting for all of *higher-order logic* (HOL).
- A labeling scheme.

Primary goal: *Develop an accepted "HOL normal form."*

[1] Formal Aspects of Computing, 1994.
[2] DEC TR, 1993.

# Motivation

Consider the following formula:

```
((forall a , b .  a = b and (exists b ,f, g.  p(b, f, g)
or f(g)=b)) or not not (forall a.  exists b.  a=b and
(p(a)(f,g(a, foo(a, b, b), a)) and (not (not true )))))
```

- Is every existential quantifier within a universally-quantified sentence?
- What is the outermost operator?

```
   forall a, b.
            a
          = b
        and exists b, f, g.
                  P( b, f, g )
                or  f( g )
                  = b
or not not forall a.
              exists b.
                       a
                     = b
                   and P( a )
                        ( f,
                          g( a, foo( a, b, b ), a ) )
                   and not not true
```

- Is every existential quantifier within a universally-quantified sentence?
- What is the outermost operator?

galois

# Desiderata

- No parentheses needed for precedence. Rather, we judiciously use line breaks and indentation.
- Combine the intuition of infix with the clarity of prefix.

  Remember: Yoda and Lisp-ers agree:

  ```
  (prefixing (operator (is intuitive)))
  ```
- Automatic sub-formula numbering to reference portions of a specification.
- A framework for automated specification clarity:
  - Automated fitting for long terms and sentences.
  - Automated definitions—i.e., `where` and `let` clauses (future work).

## Approach

We'll walk through one approach to satisfying these desiderata:

1. Functions & relations
2. Propositional logic
3. Predicate logic
4. Sub-formula numbering

In the following, we give verbatim input and output to our currently-implemented pretty-printer.

# Functions and Relations

By default, we enclose the arguments to functions and relations with parentheses, and comma-delimit (both of which are configurable). For readability, we provide a single space between arguments and parentheses.

```
f(a, b, c) = g(1,2,3)
```

```
  f( a, b, c )
= g( 1, 2, 3 )
```

# Functions and Relations

By default, we enclose the arguments to functions and relations with parentheses, and comma-delimit (both of which are configurable). For readability, we provide a single space between arguments and parentheses.

```
f(a, b, c) = g(1,2,3)
```

```
  f( a, b, c )
= g( 1, 2, 3 )
```

Of course, a function might have no arguments.

```
f() = g(a)
```

```
  f( )
= g( a )
```

# Functions and Relations

Users can configure a maximum argument length. If an argument exceeds the length, we split all arguments across lines.

```
P(reallyLongConstant, b, c)
```

```
P( reallyLongConstant,
   b,
   c )
```

In programs, we put delimiters *before* arguments for ease of editing. Here, we only care about reading, so we put delimiters after.

# Functions and Relations

Since this is HOL, a function can be an argument to another function or relation.

```
P(f(g),b,c)
```

```
P( f( g ), b, c )
```

# Functions and Relations

Since this is HOL, a function can be an argument to another function or relation.

```
P(f(g),b,c)
```

```
P( f( g ), b, c )
```

We also allow currying. We always automatically split curried arguments across lines.

```
P(a, b)(1)(42)
```

```
P( a, b )
 ( 1 )
 ( 42 )
```

# Functions and Relations

If a relation or function contains any terms that are curried, we automatically put each argument on a separate line for readability:

```
P(2, f(a)(b), 3, f(a)(b))
```

```
P( 2,
   f( a )
    ( b ),
   3,
   f( a )
    ( b ) )
```

# Functions and Relations

Deeply-nested functions become easy to parse visually.

```
f(g(f(2,3)(123456789, 1)(7,8)))(1) =
functName(anotherfunctName(1,2,3,4,5,6,7),
foo(h()(1,2,f(1,2))(3)), bar()(1))
```

```
  f( g( f( 2, 3 )
          ( 123456789, 1 )
          ( 7, 8 ) ) )
   ( 1 )
= functName( anotherfunctName( 1, 2, 3, 4, 5, 6, 7 ),
             foo( h( )
                   ( 1, 2, f( 1, 2 ) )
                   ( 3 ) ),
             bar( )
                ( 1 ) )
```

galois

# Propositional Logic

Binary operators (`and`, `or`, `implies`) are split across lines in infix, with the first argument indented by the width of the operator.

**true and false**

```
    true
and false
```

# Propositional Logic

Binary operators (`and`, `or`, `implies`) are split across lines in infix, with the first argument indented by the width of the operator.

```
true and false
```

```
    true
and false
```

```
(true or false) implies false
```

```
               true
            or false
implies false
```

# Propositional Logic

Unary operators are still parsed infix, noting indentation.

```
not (true and not false)
```

```
not     true
    and not false
```

[3]

[3] Thanks to Leslie Lamport for catching a bug in my rendering here.

# Propositional Logic

If precedence doesn't matter, we don't need to indent (to save space) and improve readability. Consider a conjunction with three conjuncts:

```
true and 1 = 2 and f(3) = g(2)
```

```
    true
and   1
    = 2
and   f( 3 )
    = g( 2 )
```

# Conditionals

An if-then-else clause can be considered to be a 3-place operator:

```
if P(a) then f() = g(a) else P(b)
```

```
if   P( a )
then   f( )
     = g( a )
else P( b )
```

# Let Expressions

Local definitions can be given with `let-in` expressions:

```
let a = if P(a) then f() = g(a) else P(b), b = forall a.
Q(a) in R(a, b)
```

```
let a = if   P( a )
        then   f( )
             = g( a )
        else P( b )
    b = forall a.
               Q( a )
in  R( a, b )
```

# Quantifiers

Following the style for binary operators in which we indent the operands the width of the operators, we similarly indent a quantified formula the width of the quantifiers.

```
forall b,a .  true
```

```
forall b, a.
       true
```

# Quantifiers

Following the style for binary operators in which we indent the operands the width of the operators, we similarly indent a quantified formula the width of the quantifiers.

```
forall b,a .  true
```

```
forall b, a.
       true
```

Of course we can nest quantifiers.

```
forall a, b.  exists c.  F(a,b,c)
```

```
forall a, b.
       exists c.
              F( a, b, c )
```

# Quantifiers

If the quantifiers are the same, we do not need to show precedence.

```
exists a, b.  exists c.  F(a,b,c)
```

```
exists a, b.
exists c.
       F( a, b, c )
```

# Quantifiers

If the quantifiers are the same, we do not need to show precedence.

```
exists a, b.  exists c.  F(a,b,c)
```

```
exists a, b.
exists c.
       F( a, b, c )
```

Sanity check: why didn't we pretty-print this as the following?

```
exists a, b, c.
       F( a, b, c )
```

Because we're just trying to syntactically-transform formulas, not semantically-transform them.

# Labeling Formulas

- Formula labels ease reference to sub-formulas. We automatically label sub-formulas.
- Basic idea:
  - How long the label is determines depth of sub-formula.
  - Magnitude of the label tells me on what “side” the sub-formula is.

# Labeling Formulas

- Think of the formula as a tree, such that operators and quantifiers are at the nodes.
- $n$-ary operators have $n$ children (we only have 1-, 2-, and 3-ary operators).
- Predicates are at the leaves.

# Labeling Formulas

We label nodes with children.

- The root is labeled with a 1.
- For a node labeled $n$ with one child, if its child has children, it is labeled $n1$.
- For a node labeled $n$ with two children,
  - if its left node has children, it is labeled $n0$.
  - if its right node has children, it is labeled $n2$.
- For a node labeled $n$ with three children, if its children have children, they're labeled $n0$, $n1$, and $n2$, respectively.

(All our operators have three or fewer children.)

# Labeling Formulas

Here's a simple formula with three binary operators.

```
true and false or (P() and Q())
```

```
   |          true
10 |      and false
1  | or       P( )
12 |      and Q( )
```

The root of the tree.

# Labeling Formulas

Here's a simple formula with three binary operators.

```
true and false or (P() and Q())
```

```
  |         true
10|     and false
1 | or      P( )
12|     and Q( )
```

Formula 1's left node.

# Labeling Formulas

Here's a simple formula with three binary operators.

```
true and false or (P() and Q())
```

```
  |         true
10|     and false
1 | or      P( )
12|     and Q( )
```

Formula 1's right node.

# Labeling Formulas

Here's a simple formula with three binary operators.

```
true and false or (P() and Q())
```

```
  |         true
10|     and false
1 | or      P( )
12|     and Q( )
```

An unlabeled leaf.

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |           true
11   |       and         P( a )
1120|             and Q( )
112  |           or false
```

The root of the tree.

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |            true
11   |        and         P( a )
1120|               and Q( )
112 |             or false
```

The root has one child.

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |              true
11   |        and          P( a )
1120|                and Q( )
112  |             or false
```

Formula 11's left child is a leaf and so is not labeled.

galois

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |              true
11   |        and          P( a )
1120|                 and Q( )
112 |              or false
```

Formula 11's right child is labeled.

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |           true
11   |      and        P( a )
1120 |             and Q( )
112  |           or false
```

Formula 112's left child is labeled.

# Labeling Formulas

Here's a slightly more complicated formula.

```
forall a.  true and ((P(a) and Q()) or false)
```

```
1    | forall a.
     |               true
11   |          and          P( a )
1120 |                   and Q( )
112  |               or false
```

Neither child of 1120 gets labeled.

# Labeling Formulas

Sometimes unary operators (`not`) and quantifier labels clash with binary operator labels, so we compute their labels but do not show them.

```
true and ((forall a.  P(a) and not Q()) or false)
```

```
    |     true
1   | and    forall a.
    |                  P( a )
1201|               and not Q( )
12  |     or false
```

Notice formula 1201 gets a label showing its a child of the quantifier (which would have label 120).

# Labeling Formulas

Labels can disambiguate intended precedence between operators with the same indentation.

```
true or (not (P() and Q()) or false)
```

```
    |     true
1   | or          P( )
120|     not and Q( )
12  | or false
```

# Labeling Formulas

We think of a `let` ... `in` ... expression as a binary operator that is distributed across the expression. Thus, we distribute the label, too.

```
let a = P(123456,78,910), b = P(f(), 12, 34, 56) in
R(a,b)
```

```
1| let a = P( 123456, 78, 910 )
 |     b = P( f( ), 12, 34, 56 )
1| in  R( a, b )
```

# Labeling Formulas

We similarly distribute a label for if-then-else expressions:

```
if (forall a.  P(a)) then (exists b.  Q(b)) else (forall c.  R(c))
```

```
1 | if   forall a.
  |             P( a )
1 | then exists b.
  |             Q( b )
1 | else forall c.
  |             R( c )
```

# Labeling Formulas

If there are sub-formulas to be labeled in an if-then-else expression, they are labeled 0, 1, and 2:

```
if a = b then b=c else c=d
```

```
1 | if      a
10|       = b
1 | then    b
11|       = c
1 | else    c
12|       = d
```

# Labeling Formulas

For let-expressions, we do not label the defining equations since (1) these are usually short (otherwise use another let expression), and they're ancillary to the formula:

```
let x = (a = b), y = (c = d), z = (x = y) in Q(x,y) and
P(x, y , z)
```

```
1 | let x =   a
  |         = b
  |     y =   c
  |         = d
  |     z =   x
  |         = y
1 | in      Q( x, y )
11|     and P( x, y, z )
```

We do label sub-formulas of `in`.

# Implementation

- I began the tool as a project to learn Haskell (inspired by Iavor Diatchki's Haskell class).
- Uses the BNF Converter (GPL) by Bj orn Bringert, Markus Forsberg, and Aarne Ranta:
  - `http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/` (google: "bnf converter").
  - Generates lexer/parser for the BNF specification.
- Most of this work results in heavy modifications to the pretty-printer and user-interface (enter `--help` to get options and usage).

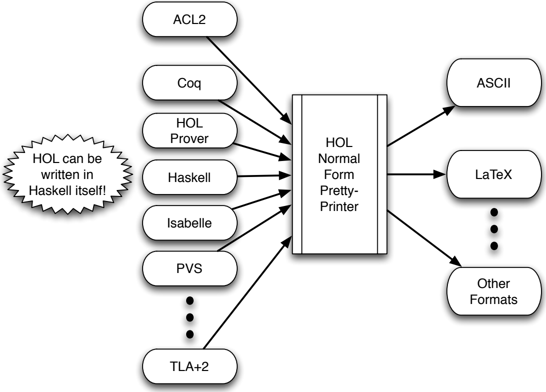
# Implementation

- I began the tool as a project to learn Haskell (inspired by Iavor Diatchki's Haskell class).
- Uses the BNF Converter (GPL) by Björn Bringert, Markus Forsberg, and Aarne Ranta:
  - `http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/` (google: "bnf converter").
  - Generates lexer/parser for the BNF specification.
- Most of this work results in heavy modifications to the pretty-printer and user-interface (enter `--help` to get options and usage).

Quick demo...

# (Intended) Usage

Most likely, take hard-to-read specs from verification tools (e.g., theorem-provers, model-checkers) and produce easier-to read specs for documentation (LaTeXand other documentation).




It's easy to modify the input and output syntax.

# (Intended) Usage

- Probably not useful for generating specs that these tools can parse themselves (most theorem-provers can't parse output in this form)—but it'd be great if this were a "standard input" in the future.
- Email me (preferably with a BNF of your favorite input language) if you have a specific input/output language you'd like pretty-printed.

# To Do/Future Work (Help Solicited!)

- Language constructs
  - Binary set-theoretic notation (e.g., $\in$, $\subseteq$, $\cup$, etc.)
  - Records & arrays
- Automatically generating `let` clauses/definitions if a formula is too large:

  "Hierarchical description or decomposition means specifying a system in terms of its pieces, specifying each of those pieces in terms of lower-level pieces, and so on. Mathematics provides a very simple, powerful mechanism for doing this: the definition" (*High-Level Specifications: Lessons from Industry*, Batson & Lamport, 2003).

galois

# Conclusions

- Your specifications are complex enough *semantically*; don't make them complex *syntactically*.

  Example: I was developing formalizations of fault-tolerant specs on a NASA project for the FAA to potentially evaluate. The specs were sometimes pages long. I had trouble parsing them sometimes. If I couldn't parse them, how could the FAA evaluate them for correctness?
- We have standard syntax for programming language specification (BNF); why not for HOL formulas? I propose the foregoing to be "HOL Normal Form."

galois