How to Pretty-Print a Long Formula

Lee Pike

Galois, Inc.
leepike@galois.com

Abstract. We propose a structured rendering for higher-order logic (HOL) to make specifications more readable. Furthermore, we present a freely-available parser and pretty-printer that implements these ideas, which we call *BeautifHOL*. We conclude by describing possible extensions to our proposal and motivate the need for formatting standards for formal specifications, generally.

1 Introduction

Three topics are sure to engender controversy: religion, politics, and this paper's subject, syntax. Despite this, my goal is to convince you that ASCII-like specifications of higher-order logic (HOL) specifications can be automatically rendered in a style more readable than the ad-hoc formatting that verificationists typically employ. To begin, consider the following formula with the usual operator precedence. The formula is written in an unstructured ASCII-like form using parentheses to show precedence:

```
(forall a, b, c . a = b and (exists b, f. P(b, c)(f) or f(b) = a)) or not
not (forall a. exists b, d, Q . a = b and (Q(a, b) and (not (not Q(b, d))))
or P(a))
```

Now quick:

– What is the outermost operator in the formula?
– Is every existential quantifier within the scope of a universal quantifier?
– What is the last disjunct of the formula?

Alternatively, consider the formula in Figure 1, and using your intuition, attempt to answer the same questions one more time. As you may have guessed, the two formulas are the same. If you found it easier to answer the questions about the latter formula, I invite you to continue reading.

1.1 Related Work

This paper pays homage to Leslie Lamport's brief and under-appreciated note entitled, "How to Write a Long Formula" [1]. Our paper emphasizes our focus on *automatically* rendering readable specifications (Lamport makes some

```
   forall a, b, c.
                a
              = b
          and exists b, f.
                        P( b, c )
                         ( f )
                     or   f( b )
                        = a
or not not forall a.
                  exists b, d, Q.
                                  a
                                = b
                            and Q( a, b )
                            and not not Q( b, d )
                         or P( a )
```

Fig. 1. Rendered Formula

recommendations whereas we have implemented a pretty-printer). Additionally, Lamport's note primarily focuses on first-order logic with set theory whereas we focus on HOL; we describe more additional differences in Section 2.5.

The major programming languages all have pretty-printers, also known as code beautifiers. Large software projects have code layout standards developers are expected to follow; for example, the open-source projects FreeBSD and GNU respectively have layout conventions [2, 3]. Indeed, pretty-printing has been important enough to the programming language communities to research and develop general-purpose pretty-printing libraries. In 1980, Oppen developed an efficient imperative pretty-printing algorithm [4]. Later, Hughes describes a general-purpose pretty-printing library in Haskell [5], which was subsequently improved upon by Wadler [6].

In general, the same attention has not been paid to the layout of specifications in the formal verification community. Most related work is in user interfaces. Research in this area includes, for example, Jape, a proof assistant in which great care is taken to make a pleasing user interface for proofs [7], and *Proof General*, which is a popular generic interface to proof assistants [8]. *ACL2s* is a plugin for the Eclipse developer environment to make using the ACL2 proof assistant easier [9].

1.2 Desiderata and Contributions

Our goal is to render HOL specifications more readable by humans. We summarize our specific desiderata as follows:

- Combine the intuition of infix notation with the clarity of prefix notation.

– Do not logically manipulate the formula—we are concerned with rendering, not deduction. We do, however, depend on the associativity of conjunction and disjunction to simplify their renderings.
– Do not depend on parentheses to clarify operator precedence (in our scheme, we judiciously use line breaks and indentation). After just a few levels of parentheses, formulas can become difficult to parse. Here we are inspired by Haskell's use of indentation to capture precedence [10].
– Provide an automated and intuitive subformula labeling scheme to ease the reference to subformulas in substitutions and proofs.
– Achieve the above without relying on special mark-up (i.e., rendering should not depend on coloration, special symbols, special fonts, etc.). Our motivation here is that we want a rendering that can be adapted for use in technical papers, text editors, and theorem-prover interfaces. We do not want the user to have to depend on special graphical interfaces.

This paper presents an approach and tool to satisfy these desiderata.

In satisfying these desiderata, the contributions in this paper include

– applying the ideas of rendering long formulas to HOL,
– publicly releasing[1] a parser and pretty-printer called *BeautifHOL* to automatically render HOL formulas in a style like in Figure 1,
– presenting a novel subformula labeling algorithm, and
– improving on Lamport's proposed layout scheme.

1.3 Outline

In Section 2, we present by example a standard rendering for HOL as well as make specific comparisons to Lamport's proposal. We have implemented the ideas presented in this paper in a tool called *BeautifHOL*, and this freely-available program is briefly described in Section 3. Possible modifications and extensions to our proposed scheme are described in Section 4, and we make concluding remarks, including our motivation for this work, in Section 5.

2 HOL Renderings

Here we present HOL renderings by example. We begin by presenting the connectives of propositional logic and then functions and relations. We then present the quantifiers. We then describe a subformula labeling scheme and conclude with specific comparisons with Lamport's proposal.

We present our ideas using a specific concrete syntax for HOL that is independent of any specific proof assistant. The inputs and outputs we present are read in and generated by the pretty-printer described in Section 3. Although we have chosen a specific concrete syntax for this presentation, both the input

[1] Available at `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/beautifHOL`. Package details are available at `\todo{TODO}`.

syntax accepted and the rendered output are configurable to accommodate different input syntaxes and output renderings (see Section 3 for more details). In the following, we show the input to our pretty-printer followed by its output in the following form:

```
input
------------------------------------------------------------
rendered output
```

Our parser ignores whitespace and line breaks in the input formula.

The renderings and the labeling scheme presented in Section 2.4 all hinge on thinking of a sentence as a tree, such that operators, including quantifiers, are nodes in the tree with one child (for negation and the quantifiers), two children (for conjunction, disjunction, and implication), or three children (for if-then-else sentences). The root node is the outermost operator of the sentence, and a node's children are its subformulas. Simple sentences (equality and relations) are at the leaves.

2.1 Propositional Logic

We begin with the standard propositional operators: conjunction, disjunction, and implication, and negation.

The Binary Operators Let us begin with conjunction. We indent the operands to show they are within the scope of the operator.

```
P(a) and P(b)
------------------------------------------------------------
    P( a )
and P( b )
```

Notice that despite being infix notation, we get the clarity of prefix notation. We can use this scheme for the binary operators disjunction and implication, too. This rendering is applied recursively, so outermost operators are rendered as leftmost operators. For example, consider the rendering of a formula in which the first operator in the formula has precedence.

```
(P(a) or P(b)) implies P(c)
------------------------------------------------------------
           P( a )
        or P( b )
implies P( c )
```

Now consider the rendering if the second operator has precedence. In both cases, the formula is rendered infix and we do not require parentheses to show precedence.

```
P(a) or (P(b) implies P(c))
--------------------------------
   P( a )
or         P( b )
   implies P( c )
```

Furthermore, like in Lamport's scheme, if we are rendering a series of associative operators, we can save space and improve readability by not indenting. Consider the following conjunction:

```
P(a) and (P(b) or Q(b)) and P(f(3))
--------------------------------
    P( a )
and    P( b )
    or Q( b )
and P( f( 3 ) )
```

Unary Operators The scope of negation can also be shown by spacing and indentation. For a unary operator like negation, there is no need to place the single operand on a new line. Consider the following example:

```
not not P(3)
--------------------------------
not not P( 3 )
```

Here is an example in which we negate a binary operator:

```
not (P(a) and not P(b))
--------------------------------
not     P( a )
    and not P( b )
```

Other Expressions When specifying programs in HOL, if-then-else and let-expressions are convenient. If-then-else expressions can be rendered like a three-place operator. For example,

```
if P(a) then (P(b) and P(c)) else (P(b) or P(c))
--------------------------------
if   P( a )
then     P( b )
     and P( c )
else    P( b )
     or P( c )
```

We provide a conventional rendering for let-in sentences, similar to what one finds in a programming language like Haskell [10].

```
let a = if P(y) then Q(f(y)) else P(y), b = Q(x) in R(a, b)
```

```
let a = if   P( y )
        then Q( f( y ) )
        else P( y )
    b = Q( x )
in  R( a, b )
```

2.2 Functions and Relations

We have already seen some simple examples containing simple predicates above. Functions can be hard to visually parse when they contain a lot of parameters, their parameters have long names, or the parameters are functions, which containing parameters themselves. In the implementation of *BeautifHOL*, if some parameter exceeds some user-defined maximum number of characters, it automatically places parameters on separate lines.

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

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

In the following, the function `f` together with its parameters exceeds our user-defined limit of characters:

```
P(g(a), b, f(a, b, c, d, e, f, g, h))
```

```
P( g( a ),
   b,
   f( a, b, c, d, e, f, g, h ) )
```

Noting these two examples, consider that in programs, we put delimiters before parameters for ease of editing—for example, when specifying constructors in a datatype. Here, we are concerned with reading them rather than editing them, so we put delimiters afterwards, as is conventional in mathematics. We also prefer to place a space between parentheses and the list of parameters.

In HOL, functions are often curried. We automatically place curried parameters on separate lines for ease of reading.

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

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

We treat equality as an infix binary operator, like conjunction. Thus, we place its operands on separate lines and past the function identifier to show scope with indentation. This style works especially well if the operands are themselves large.

```
f(a, b, c, d) = g(1, 2, 3) and h(42, a, b, c) = h(pi, a, b, c, d)
```

```
      f( a, b, c, d )
    = g( 1, 2, 3 )
and   h( 42, a, b, c )
    = h( pi, a, b, c, d )
```

Complex specifications of 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 ) )
```

2.3 Quantifiers

We place the sentence that is quantified on a new line, indenting past the quantifier.

```
forall b, P. P(b)
```

```
forall b, P.
       P( b )
```

We place quantified sentences on a new line (rather than just indenting, like with negation) for two reasons. First, it is not unusual to have deeply nested quantifiers in HOL specifications, and indentation improves readability (deeply nested negations are more rare, since negation elimination is often applied immediately). Second, we describe a labeling scheme in Section 2.4 that assigns labels to subformulas. One often wishes to refer to a sentence quantified over (for example, when referring to a sentence with instantiated variables). By placing the quantified sentence on its own line, we can render labels on the same line as the sentence. Here is an example of nested quantifiers:

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

For readability and to save space, we do not to indent the quantified sentence if the quantifiers are the same (similar to conjunction and disjunction).

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

2.4 Labeling Subformulas

For large formulas, labeling subformulas allows one to easily refer to a portion of a formula. Here, we present a novel but simple labeling algorithm that assigns labels which describe the structure of the formula.

More specifically, our approach encodes the tree structure of a sentence that was noted at the outset of Section 2, . From a node's label, one can determine the node's depth in the tree, what operand is at its parent node, its grandparent node, and so on, and which branches from its ancestor nodes are followed to reach the subformula. We propose that labels with meaning can help one maintain context more easily in long complex proofs.

The labeling algorithm we propose is as follows. We assign natural numbers to the nodes according to the following rules. First, the root node is always labeled 1. Now suppose that a node A has label n and has one or more children. Then the children of n are labeled as follows. First, if a child is a leaf, it is unlabeled. Otherwise, if a child is not a leaf, it is labeled according to the following rules:

1. If A is a negated sentence, then its child is labeled $10n + 0$.
2. If A is a conjunction, then
   - the left child is labeled $10n + 1$, and
   - the right child is labeled $10n + 2$.
3. If A is a disjunction or implication, then
   - the left child is labeled $10n + 3$, and
   - the right child is labeled $10n + 4$.
4. If A is an if-then-else sentence, then
   - the first child is labeled $10n + 5$,
   - the second child is labeled $10n + 6$, and
   - the first child is labeled $10n + 7$,
5. If A is a universally-quantified sentence, then its child is labeled $10n + 8$.
6. If A is an existentially-quantified sentence, then its child is labeled $10n + 9$.
7. Otherwise, its children inherit the label n.

Consider some examples of formulas rendered and labeled by *BeautifHOL*.[2]

```
(P(a) and P(b)) or (Q(a, b) and Q(b, c))
```

```
  |        P( a )
13|    and P( b )
1 | or     Q( a, b )
14|    and Q( b, c )
```

The outermost node is labeled 1. Because it is a disjunction, its left child (`(P(a) and P(b))`) is labeled $10 \times 1 + 3 = 13$, and its right child (`(Q(a, b) and Q(b, c))`) is labeled $10 \times 1 + 4 = 14$. The other subformulas are not labeled, since they are leaves. From a label alone, one can deduce the context. For example, in the above sentence, from the label 13, one can deduce that the formula `P(a) and P(b)` is the left operand, and disjunction is the outermost operator of the formula.

In the next formula, notice that the label for the if-then-else sentence is "distributed" across the formula. So from the label 16, we know the subformula `1 = f(a, b)` is in the context of the `then` branch.

```
if P(a) then 1 = f(a, b) else P(c) and Q(b)
```

```
1 | if   P( a )
1 | then   1
16|      = f( a, b )
1 | else     P( c )
17|      and Q( b )
```

Sometimes printing two labels will lead to multiple labels clashing on the same line. For example, consider the following labeled formula:

```
not (forall a. P(a) and Q(a))
```

```
1  | not forall a.
   |               P( a )
108|           and Q( a )
```

We do not print the label for the subformula `forall a. ...` (with label 10) since it would clash with the label 1 for the full formula. However, the subformula `P(a) and Q(a)` is labeled 108, showing that it is the subformula of negated universally-quantified sentence. If there is a conflict in labels, we show the label associated with the outermost operator in conflict. Only negated subformulas and quantified subformulas have potentially conflicted labels.

[2] Subformulas are labeled by default in *BeautifHOL*, but they can be suppressed using the flag `--nolabels`.

2.5 Comparison to Lamport's Proposal

As we mentioned in the outset, much of our proposal is inspired by Lamport's original proposal for rendering first-order logic. Here we briefly look at some key differences.

First, regarding binary operators, Lamport proposes to place each conjunct on a new line and prefix each conjunct with the operator [1]:

```
and P( a )
and P( b )
and P( c )
```

However, doing so is both unnecessary and not strictly infix. Our proposal breaks operands across lines, but keeps infix notation. In particular, regarding the implication operator, Lamport writes, "I have not found a good general method of writing $A \Rightarrow B$ when A and B are long formulas" [1]. The problem is that Lamport's scheme would have us prefix each operand with an implication operator, which does not work for non-associative operators, as he himself points out. That is, if we treated implication like conjunction or disjunction under Lamport's scheme, for `(P(a) implies P(b)) implies P(c)`, we would write something like the following:

```
implies P(a)
implies P(b)
implies P(c)
```

However, without parentheses, this could be interpreted either as `P(a) implies (P(b) implies P(c))` or `(P(a) implies (P(b)) implies P(c)`. Consequently, Lamport suggests writing implication like

```
and P(a)
and P(b)
implies P(c)
```

which only reads well if the antecedent is small, as Lamport himself also notes. Our choice to render implication infix and indent the antecedent just like the consequent solves these problems. Still, with a large antecedent, Lamport believes our choice still does not read well.[3] We believe the issue turns on whether one takes the perspective of viewing a rendered formula primarily as a tree, the root of which is on the left and the leaves are on the right, or as a formula to be read from top to bottom. For example, recall Figure 1: for the outermost operator (disjunction), do you begin by reading the operator (`or`) or its first operand (`forall a, b, c. ...`)? With the first of these perspectives (the one we take), the size of the antecedent is irrelevant.

[3] Private correspondence (January 2009).

Finally, regarding our labeling mechanism, Lamport proposes a different approach [1]. Lamport focuses on labeling conjuncts and disjuncts (other subformulas are not labeled). He assigns numbers to conjuncts and letters to disjuncts. Furthermore, he allows variable substitution in universally-quantified or existentially-quantified formulas to be part of the label. So, for example, $1.c(q).2$ might represent the subformula in the second conjunct of the second disjunct, where we have substituted in q in the quantified formula, in the first conjunct. For the language of TLA^{+2}, Lamport extends his original scheme [11].

3 Implementation

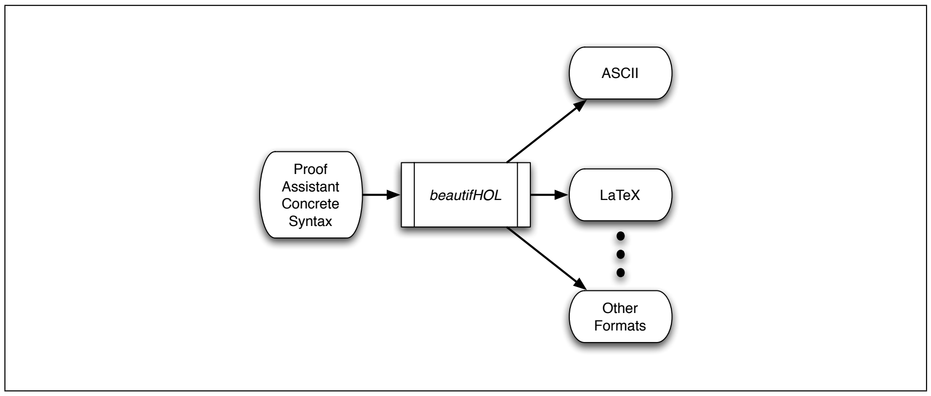

Fig. 2. Pretty-Printer Framework

Our pretty-printer *BeautifHOL* takes in a HOL formula and pretty-prints it as described above. In the current implementation, the concrete syntax of our input and output formulas corresponds to the input and output formulas shown in this paper (the parser ignores whitespace and line breaks in input formulas). However, the concrete syntax of the input formulas accepted by *BeautifHOL* can be modified by changing a labeled Backus Normal Form (BNF) file, yielding a family of pretty-printers. Additionally, there is a configuration file to modify the concrete syntax of the output to yield various representations, including representations in ASCII, LaTeXsource, and so on. We illustrate this in Figure 2.

Individual formulas can be input at the command line, or files containing formulas can be read in. Options include suppressing output (i.e., just signaling whether parsing is successful), suppressing the parse tree, outputting the input formula together with its rendering (for debugging), and suppressing subformula labeling.

The lexer, parser, and language specification for our pretty-printer is generated from *BNF Converter*, authored by Björn Bringert, Markus Forsberg, and

Aarne Ranta.[4] *BeautifHOL* is implemented in Haskell (as is *BNF Converter*), and is released under a BSD3 license. The source code is available on Hackage.[1] The pretty-printer is alpha-level software, the purpose of which is to demonstrate the ideas presented in this paper. Contributions are welcomed!

4 Extensions and Future Work

Here we describe possible extensions to the *BeautifHOL* pretty-printer and future work for rendering HOL specifications.

Operator Delimiters Inspired by the Programatica graphical user interface in which delimiters are used to show the scope of a definition [12], Magnus Carlsson suggests[5] the use of delimiters to show the scope of binary operators, such as the following hypothesized rendering:

```
(P(a) and (not Q(b, b) implies (P(c) or Q(a, b)))) or (P(c) implies
(Q(a, c) and (P(b) or Q(b, a))))
```

```
        |P( a )
   |and |        |not Q( b, b )
   |    |implies |   |P( c )
   |             |or |Q( a, b )
or |        |P( c )
   |implies |    |Q( a, c )
            |and |   |P( b )
                 |or |Q( b, a )
```

Delimiters are particularly beneficial when operands are large.

"Best Fit" for Relations and Functions Another extension would be to *automatically* decompose specifications by introducing local definitions (i.e., let-in expressions) as needed to make a formula more readable, particularly if a formula requires line wrapping to be displayed. For example, suppose the line width is 80 characters, and a formula is the following:

```
    ...
and ... f ( a, b, ... c )
    ...
```

such that the length of the line containing the function `f`, exceeds 80 characters. The pretty-printer could introduce a local definition to reduce the lines length:

[4] The tool, released under a GPL license, and documentation can be downloaded at `http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/`.

[5] Via personal communication (September, 2008).

```
let f' = f ( a, b, ... c )
in      ...
   and ... f'
        ...
```

Let expressions can be introduced iteratively until the formula does not require line wrapping.

Another extension would be to have more control over how functions are rendered. Depending on user constraints on a formula's length and width, a formula might be rendered differently depending on its context within a formula. For example, the following rendering minimizes the length (e.g., number of lines) of a relation:

```
P( a, b, f( c, d, e ), g( f, h( a, b, c, d, e ) ), j, k )
```

To minimize the width (e.g., number of characters in a line) of the same relation, we choose a different rendering:

```
P( a,
   b,
   f( c, d, e ),
   g( f,
      h( a,
         b,
         c,
         d,
         e ) ),
   j,
   k )
```

The following is a compromise between the two:

```
P( a, b,
   f( c, d, e ),
   g( f, h( a, b, c, d, e ) ),
   j, k )
```

Pretty-printer combinators developed by Hughes are designed specifically to address these layout problems [5].

Additional Syntax The simplest modification is to add new syntax. For example, binary operators from algebra (e.g., $\leq$, $+$, etc.) as well as set-theoretic operators (e.g., $\in$, $\subseteq$, $\cup$, etc.) are both often found in HOL specifications. In general, these infix binary operators can be rendered like we render equality. As needed, syntax for records, arrays, and lists can be added. Adding new syntax requires modifying the labeled BNF grammar for input formulas as well as proving a new pretty-printing rule for the construct, if no rule is reused.

5 Conclusion

While I was a member of the NASA Langley Research Center Formal Methods Group, I worked on the SPIDER project.[6] The portion of the project I was involved in was the formal analysis of fault-tolerant communication protocols, mainly carried out in a mechanical theorem-prover. Our team consisted of about four individuals, plus intermittent visitors, concurrently developing specifications and proofs. Even among this small team, I was surprised to see a broad range of specification styles. Furthermore, I found myself spending precious time simply parsing large formulas—even ones I wrote—particular in the midst of deep proofs. I was thinking about how to parse formulas instead of how to finish proofs!

I believe the difficulties associated with readable specifications are even more problematic for the formal verification community than the programming community for three reasons. First, programming projects are often larger—orders-of-magnitude larger—than formal specification and verification projects. The size of programming communities compels project leaders to institute code formatting standards. I am not aware of any formal project in which specification standards are explicitly issued and enforced (i.e., repository commits are not accepted unless the standard is adhered to). Second, because of the size of these communities, programmer tools are more mature: most major programming languages have pretty-printers available for them, making adherence to the standards easier. Third, specifications are not only written but formally reasoned about. Rewriting formulas in proofs can dramatically increase their size and syntactic complexity beyond the original syntactic complexity of the specification. These rewrite expansions do not normally occur in programming (perhaps with the exception of macros output). Pretty-printing research may seem mundane or irrelevant. Nevertheless, user-centric issues like pretty-printing are important as formal specifications become more central to the software development process and have users and reviewers from outside the community.

My goal in this paper has been to make present a proposal for rendering HOL specifications and provide a tool for doing so. Syntax, of course, is a matter of taste, and I expect additional modifications and improvements will be made to the proposals I have made. More generally, I wished to (re)introduce Lamport's original paper to the formal verification community. The problem of rendering formal specifications to be more readable is an important one, and good solutions will hasten the adoption formal methods amongst the wider Computer Science community.

---

[6]

Comment: Get website

Acknowledgments

In addition to the comments Leslie Lamport provided to this author, noted throughout the paper, Lamport reviewed a preliminary draft of this work and caught a small bug in my rendering of negation (communicated in private correspondence). I received numerous constructive comments at the Galois Technical Seminar, particularly from Iavor Diatchki, Joe Hurd, and John Matthews. Alwyn Goodloe commented on an early draft. In particular, Magnus Carlsson provided enormously detailed and helpful comments.

References

1. Lamport, L.: How to write a long formula (short communication). Formal Aspects of Computing **6**(5) (1994) 580–584
2. Free Software Foundation: style – kernel source file style guide. FreeBSD Kernel Developer's Manual Retreived Dec. 2008. Available at `http://www.freebsd.org/cgi/man.cgi?query=style&sektion=9`.
3. Free Software Foundation: GNU coding standards. Webpage Retreived Dec. 2008. Available at `http://www.gnu.org/prep/standards/`.
4. Oppen, D.C.: Prettyprinting. ACM Transactions on Programming Languages and Systems **2**(4) (1980) 465–483
5. Hughes, J.: The design of a pretty-printing library. In: Advanced Functional Programming, Springer Verlag (1995) 53–96
6. Wadler, P.: A prettier printer. In: Journal of Functional Programming. (1998) 223–244
7. Bornat, R., Sufrin, B.: A minimal graphical user interface for the jape proof calculator. Formal Aspects of Computing **11**(3) (1999) 244–271
8. Aspinall, D.: Proof general: A generic tool for proof development. In: TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2000) 38–42
9. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: Acl2s: "the acl2 sedan". In: ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society (2007) 59–60
10. Jones, S.P.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (May 2003)
11. Lamport, L.: TLA^{+2}, A Preliminary Guide. (April 2008) Available at `http://research.microsoft.com/en-us/um/people/lamport/tla/tla2-guide.pdf`.
12. Hallgren, T.: Haskell tools from the programatica project. In: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, ACM (2003) 103–106 Available at `http://ogi.altocumulus.org/~hallgren/Programatica/HW2003/`.