119

How to Write a Long Formula

Leslie Lamport

December 25, 1993
Minor correction: January 18, 1994

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

How to Write a Long Formula

Leslie Lamport

December 25, 1993
Minor correction: January 18, 1994

Author's Abstract

Standard mathematical notation works well for short formulas, but not for the longer ones often written by computer scientists. Notations are proposed to make one or two-page formulas easier to read and reason about.

Introduction

Mathematicians seldom write formulas longer than a dozen or so lines. Computer scientists often write much longer formulas. For example, an invariant of a concurrent algorithm can occupy more than a page, and the specification of a real system can be a formula dozens or even hundreds of pages long. Standard mathematical notation works well for short formulas, but not for long ones. I propose a few simple notations for writing formulas of up to a couple of pages. These notations can make formulas much easier to read and reason about.

Formulas significantly longer than two pages require hierarchical structuring. Methods for structuring long programs can be used to structure long formulas. Programs of less than a dozen or so pages can be adequately structured with procedures; longer programs require some method of grouping procedures into modules. The definition is the mathematical analog of the procedure. Definitions suffice for structuring formulas of up to about a dozen pages. For longer formulas, some form of module structure is also needed.

Any formula can be written with a hierarchy of definitions, each only a few lines long. However, just as programs become hard to read if broken into too many procedures, formulas are hard to read if broken into definitions that are too small. In my experience, the best way to structure a long formula is in terms of individual formulas of up to a page or two.

Writing Formulas

Consider the following definition, written with standard mathematical conventions. (The examples come from the invariant of an unpublished correctness proof for a cache coherence algorithm; the reader is not expected to understand them.)

$$\begin{aligned}
\mathit{memQLoc}(a) &\equiv \begin{cases} \text{“None”} & \text{if } \mathit{Locs} = \emptyset \\ \max(\mathit{Locs}) & \text{otherwise} \end{cases} \\
\text{where } \mathit{Locs} &\equiv \{ i \in \{0 \ldots |\mathit{memQ}| - 1\} : \\
&\qquad (\mathit{memQ}[i].\mathit{req}.\mathit{type} = \text{“Write”}) \\
&\qquad\quad \wedge\ (\mathit{memQ}[i].\mathit{req}.\mathit{adr} = a) \quad \}
\end{aligned}$$

This definition is easy to read because it is short. However, suppose that “None” and max(*Locs*) were replaced by much longer expressions. We would then see that the “where” construct is bad because it forces us to read the entire definition of $memQLoc(a)$ before we learn what $Locs$ is. A structure that scales better to large formulas is

$$\begin{array}{lll}\textbf{let} & Locs \equiv & \{i \in \{0 \ldots |memQ| - 1\} : \\ & & \quad (memQ[i].req.type = \text{“Write”}) \\ & & \quad\quad \wedge\ (memQ[i].req.adr = a) \quad \} \\ \textbf{in} & memQLoc(a) \equiv & \begin{cases} \text{“None”} & \text{if } Locs = \emptyset \\ \max(Locs) & \text{otherwise} \end{cases}\end{array}$$

Suppose once again that “None” were replaced by a long expression e, perhaps crossing onto the next page. The typographic difficulties posed by the resulting large left brace are daunting. Simply removing the brace still leaves us with the problem of where to put the condition $Locs = \emptyset$. If it goes after e, we have to read several lines before discovering the structure of the definition. If it goes at the end of the first line, we read the $Locs = \emptyset$ in the middle of reading e. A better notation is the **if/then/else** construct used in programming languages.

$$\begin{array}{lll}\textbf{let} & Locs \equiv & \{i \in \{0 \ldots |memQ| - 1\} : \\ & & \quad (memQ[i].req.type = \text{“Write”}) \\ & & \quad\quad \wedge\ (memQ[i].req.adr = a) \quad \} \\ \textbf{in} & memQLoc(a) \equiv & \textbf{if } Locs = \emptyset \textbf{ then } \text{“None”} \\ & & \qquad\qquad\quad \textbf{else } \max(Locs)\end{array}$$

The **if/then/else** makes the structure immediately clear, even for long formulas. The obvious analog of the **case** construct of programming languages works for definitions with more than two alternatives. The customary closing **end** (or **fi**) is unnecessary, because we can use parentheses and indentation to delimit the scope of an **if** or **case**.

The original version of the definition had an important feature that has been lost in these transformations: we could see at once that it was a definition of $memQLoc(a)$. One further change recovers this feature.

$$\begin{array}{ll} memQLoc(a) \equiv & \textbf{let } Locs \equiv \{i \in \{0 \ldots |memQ| - 1\} : \\ & \qquad\qquad\qquad (memQ[i].req.type = \text{“Write”}) \\ & \qquad\qquad\qquad\quad \wedge\ (memQ[i].req.adr = a) \quad \} \\ & \textbf{in if } Locs = \emptyset \textbf{ then } \text{“None”} \\ & \qquad\qquad\qquad \textbf{else } \max(Locs)\end{array}$$

The basic problem with the “if … otherwise” construct is shared by all infix operators: we discover the high-level structure only after reading to the end of the first argument. Consider the following formula.

$$\begin{array}{l}
(\forall\, c \in \mathit{CacheAddress} : \\
\quad \mathit{cache}[p, c] \in ([\![\mathit{adr} : \mathit{Address}, \mathit{val} : \mathit{Value}]\!] \cup \{\text{“Invalid”}\})) \\
\quad \wedge\ ((\mathit{request}[p] \in \mathit{Request}) \\
\qquad \vee\ ((\mathit{request}[p] = \text{“Ready”}) \wedge (\mathit{state}[p] = \text{“Idle”}))) \\
\quad \wedge\ (\mathit{response}[p] \in \mathit{Value})
\end{array}$$

We have to read to the end of the second line, and count parentheses, before learning that the formula is a conjunction. One possible solution is prefix notation, writing $\wedge(A,\, B,\, C)$ instead of $A \wedge B \wedge C$.

$$\begin{array}{l}
\wedge\ (\forall\, c \in \mathit{CacheAddress} : \\
\qquad \mathit{cache}[p, c] \in ([\![\mathit{adr} : \mathit{Address}, \mathit{val} : \mathit{Value}]\!] \cup \{\text{“Invalid”}\}), \\
\quad \vee\ (\mathit{request}[p] \in \mathit{Request}, \\
\qquad \wedge\ (\mathit{request}[p] = \text{“Ready”}, \\
\qquad\quad \mathit{state}[p] = \text{“Idle”})), \\
\quad \mathit{response}[p] \in \mathit{Value})
\end{array}$$

This formula is easy to read only because of the way it is indented. If one needs indentation anyway, why not use it to eliminate the parentheses and commas required by a prefix notation? We write the formula $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ as the aligned list

$$\begin{array}{l}
\wedge\ A_1 \\
\wedge\ A_2 \\
\quad \ldots \\
\wedge\ A_n
\end{array}$$

and write disjunctions similarly. We can then use indentation to eliminate parentheses, writing the formula above as

$$\begin{array}{l}
\wedge\ \forall\, c \in \mathit{CacheAddress} : \\
\qquad \mathit{cache}[p, c] \in ([\![\mathit{adr} : \mathit{Address}, \mathit{val} : \mathit{Value}]\!] \cup \{\text{“Invalid”}\}) \\
\wedge\ \vee\ \mathit{request}[p] \in \mathit{Request} \\
\quad \vee\ \wedge\ \mathit{request}[p] = \text{“Ready”} \\
\qquad \wedge\ \mathit{state}[p] = \text{“Idle”} \\
\wedge\ \mathit{response}[p] \in \mathit{Value}
\end{array}$$

We continue to use $\wedge$ and $\vee$ as infix operators in subformulas. For example, the second conjunct of this formula can also be written

$$\begin{array}{l}
\wedge\ \vee\ \mathit{request}[p] \in \mathit{Request} \\
\quad \vee\ (\mathit{request}[p] = \text{“Ready”}) \wedge (\mathit{state}[p] = \text{“Idle”})
\end{array}$$

The list convention for conjunction and disjunction can be used for other associative operators, including addition and multiplication. However, it does not work for the nonassociative boolean operator $\Rightarrow$ (implies). I have not found a good general method of writing $A \Rightarrow B$ when A and B are long formulas. When A and B are conjunctions or disjunctions, the format

$$\begin{array}{l} \wedge\ A_1 \\ \quad \ldots \\ \wedge\ A_m \\ \Rightarrow \begin{array}{l} \wedge\ B_1 \\ \quad \ldots \\ \wedge\ B_n \end{array} \end{array}$$

works fairly well if $A_1 \wedge \ldots \wedge A_m$ is only a few lines long.

Writing conjunctions and disjunctions as lists lets us take full advantage of indentation to eliminate parentheses. Indentation has meaning; shifting an expression to the left or right changes the way a formula is parsed. It is not hard to devise precise rules for parsing these two-dimensional formulas. However, there is some question about what formulas should be allowed. For example, should it be legal to write $(A_1 \vee A_2) \wedge B$ as follows?

$$\begin{array}{l} \quad \vee\ A_1 \\ \quad \vee\ A_2 \\ \wedge\ B \end{array}$$

Answers to these questions will evolve as people use the notation.

Numbering Parts of Formulas

We don't just write formulas, we also reason about them. Reasoning about a large formula requires a convenient way of referring to its components. With the list convention, we can name individual conjuncts and disjuncts by numbering them. The ith conjunct or disjunct of a formula named F is called $F.i$. A universally quantified formula can be viewed as a conjunction, where the yth conjunct of $\forall x : Q$ is $Q[y/x]$, the formula obtained by substituting y for x in Q. If F is the name of the formula $\forall x : Q$, then we take $F(y)$ to be the name of the formula $Q[y/x]$. A similar convention applies to existential quantification.

Figure 1 illustrates the use of these structuring and naming conventions in a real example—the definition of an invariant I for a cache coherence algorithm. For simplicity, only the outermost three levels of con-

```
$I$ $\equiv$ let $cacheLocs(p, a) \equiv \{c \in CacheAddress : \land\ cache[p, c] \neq \text{“Invalid”}$
                                                       $\land\ cache[p, c].adr = a \quad \}$
        $inCache(p, a) \equiv cacheLocs(p, a) \neq \emptyset$
        $memQLoc(a) \equiv$ let $Locs \equiv \{i \in \{0 \ldots |memQ| - 1\} :$
                                        $\land\ memQ[i].req.type = \text{“Write”}$
                                        $\land\ memQ[i].req.adr = a \quad \}$
                          in if $Locs = \emptyset$ then “None”
                                             else $\max(Locs)$
        $memVal(a) \equiv$ if $memQLoc(a) = \text{“None”}$
                          then $mainMemory[a]$
                          else $memQ[memQLoc(a)].req.val$
    in  1.$\land\ \forall\, p \in Process$ :
            1.$\land\ \forall\, a \in Address$ :
                1.$\land\ \#cacheLocs(p, a) \leq 1$
                2.$\land\ inCache(p, a) \Rightarrow (cacheVal(p, a) = memVal(a))$
                3.$\land\ mainMemory[a] \in Value$
            2.$\land\ \forall\, c \in CacheAddress$ :
                $cache[p, c] \in ([\![adr : Address,\ val : Value]\!] \cup \{\text{“Invalid”}\})$
            3.$\land$ a.$\lor\ request[p] \in Request$
                b.$\lor\ \land\ request[p] = \text{“Ready”}$
                     $\land\ state[p] = \text{“Idle”}$
            4.$\land\ response[p] \in Value$
            5.$\land$ 1.$\land\ state[p] \in \{\text{“RdCache”}, \text{“MemWait”},$
                                  $\text{“BusWait”}, \text{“WrDone”}, \text{“Idle”}\}$
                2.$\land\ (state[p] = \text{“RdCache”}) \Rightarrow \land\ request[p].type = \text{“Read”}$
                                                  $\land\ inCache(p, request[p].adr)$
                3.$\land\ (state[p] = \text{“MemWait”})$
                     $\Rightarrow \land\ \neg inCache(p, request[p].adr)$
                        $\land\ \#\{i \in \{0 \ldots |memQ| - 1\} :$
                                 $\land\ p = memQ[i].proc$
                                 $\land\ memQ[i].req.type = \text{“Read”}\} = 1$
                4.$\land\ (state[p] = \text{“BusWait”}) \land (request[p].type = \text{“Read”})$
                     $\Rightarrow \neg inCache(p, request[p].adr)$
                5.$\land\ (state[p] = \text{“WrDone”}) \Rightarrow (request[p].type = \text{“Write”})$
        2.$\land\ memQ \in SequenceOf([\![proc : Process,\ req : Request]\!])$
        3.$\land\ \forall\, i \in \{0 \ldots |memQ| - 1\}$ :
            $memQ[i].req.type = \text{“Read”}$
              $\Rightarrow$ 1.$\land\ state[memQ[i].proc] = \text{“MemWait”}$
                 2.$\land\ request[memQ[i].proc] = memQ[i].req$
```

Figure 1: An invariant of a cache coherence algorithm.

juncts and disjuncts are labeled. (I like to label conjuncts with numbers and disjuncts with letters.) The naming convention implies that $I.2$ is the formula $memQ \in SequenceOf(\ldots)$, and $I.1(q).3.\mathrm{a}$ is the formula $request[q] \in Request$.

Conclusion

The notations introduced here will be unfamiliar to most readers, and unfamiliar notation usually seems unnatural. I have used the notations for several years, and I now find them indispensable. I urge the reader to rewrite formula I of Figure 1 in conventional notation and compare it with the original. Having to keep track of six or seven levels of parentheses reveals the advantage of using indentation to eliminate parentheses.