THE AUTOMATED PROOF OF A TRACE TRANSFORMATION FOR A BITONIC SORT

Chua-Huang Huang & Christian Lengauer

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-30 October 1984

Abstract

In his third volume, Knuth presents Batcher's bitonic sort as a sorting network. With concurrency, this sorting network can be executed in logarithmic time. Knuth suggests a formal argument for the correctness of the bitonic sorting algorithm (as an exercise), but addresses the question of concurrency only informally.

We develop a program for the bitonic sort by

(1) deriving a stepwise refinement from Knuth's informal description of the algorithm,

(2) deriving from the refinement a sequential execution or "trace" of order O(n log n) in the length n of the sequence to be sorted, and

(3) transforming the sequential trace into a parallel trace of order O(log n) while preserving its correctness.

We shall be informal in Steps 1 and 2 - although these steps can be formalized. But we will provide a formal treatment of Step 3 and report on the certification of this treatment in a mechanized logic.

This work is a contribution to

(a) the optimization of programs (via concurrency) through transformation, and

(b) the mechanized treatment of formal program derivations.

1. Introduction

A sequence $a = \langle a_0, a_1, \ldots, a_n \rangle$ is in *bitonic order* if $a_0 \geq \ldots \geq a_i \leq \ldots \leq a_n$ for some i such that $0 \leq i \leq n$. The bitonic sorting algorithm sorts a sequence $\langle a_0, a_1, \ldots, a_n \rangle$ that is already in bitonic order into ascending order by sorting the subsequences $\langle a_0, a_2, \ldots \rangle$ and $\langle a_1, a_3, \ldots \rangle$ independently and then comparing and, if necessary, interchanging (a_0, a_1), (a_2, a_3), ... Since the subsequences of a bitonic sequence are also bitonic, $\langle a_0, a_2, \ldots \rangle$ and $\langle a_1, a_3, \ldots \rangle$ can be sorted by the same algorithm, until all subsequences have length 1.

Knuth [4] presents the bitonic sort as a *sorting network*. A node in a sorting network is a *comparator module* which takes two (not necessarily adjacent) sequence elements as inputs, compares them and, if necessary, interchanges them into ascending order. The bitonic sort can sort a bitonically ordered sequence of length n in O(log n) time, if comparator modules may be applied concurrently.

The significance of the bitonic sorting algorithm lies in the fact that we can derive from it a network that sorts arbitrary (not bitonic) sequences.[1] We have to make the following additional requirements:

(1) the length of the sequence is a power of 2, and

(2) two versions of the comparator module are available, one that swaps into ascending order and one that swaps into descending order.

Then, a sorting network based on the bitonic sort exists that sorts sequence a in $O(\log^2 n)$ time if comparator modules may be applied concurrently. Each node of the network appropriately represents one or the other type of comparator module. In the following, we shall only deal with the original bitonic sort. The extension to general sequences does not add any new issue in our program development.

Knuth suggests the *zero-one principle* to prove the correctness of the bitonic sort,[2] but does not deal formally with concurrency. We apply a programming methodology which can deal formally with concurrency [5]. In this methodology, the derivation of concurrency proceeds by a successive compression of the program's executions based on the declaration of certain useful program properties. We call program executions *traces*.

We are interested in the mechanical certification of trace transformations. As does the bitonic sort, most interesting programs contain recursions or loops. The most effective and practical transformations of their traces will also be recursive, and their proofs of correctness will require induction. We are therefore interested in the mechanical treatment of recursion and induction. We use a powerful induction prover [2] that is based on a mechanized functional logic particularly suitable for program verification [1]. The

[1]See Exercise 13 of [4].

[2]See Exercise 10 of [4].

prover is designed to prove theorems about recursive functions but is not an expert on sorting networks and their trace transformations. We must "teach" it the theory of trace transformations before it is able to certify the transformation of the bitonic sort.

The following section describes the bitonic sort and explains its trace transformation informally. Sect. 3 presents our implementation of the trace transformation theory of sorting networks in the mechanized logic and the proof of the trace transformation theorem of the bitonic sort on the mechanical prover. In the concluding section, we evaluate the approach of the mechanized certification of semantic properties. The proof session in which our theorem of the bitonic sort was certified is presented in an appendix.

2. The Bitonic Sort

We refine the bitonic sort in a special-purpose language: the language of *sorting networks*. A refinement consists of a *name* (with an optional list of formal parameters) separated by a colon from a *body*. There are three choices of refinement body:

(1) The *null statement* `skip` does nothing.

(2) The *comparator module* `cs(i,j)` accesses a sequence `a` of numbers. It compares elements a_i and a_j and, if necessary, interchanges them into order.

(3) The *composition* `S1;S2` of refinements `S1` and `S2` applies `S2` to the results of `S1`. Each of `S1` and `S2` can be a refinement call (i.e., a refinement name, maybe, with an actual parameter list), a comparator module, or the null statement. Sequences of compositions `S1;S2;...;Sn` are also permitted. Refinement calls may be recursive.

Composition may be implemented by execution in order but need not be in all cases. For instance, in a programming language with assignment statements, the two assignments `x:=x+a; x:=x+b` could also be executed in reverse order, and the two assignments `x:=3; y:=5` could also be executed in parallel. We prefer to think of a refinement as a mathematically defined object that does not address questions of execution. We shall deal with the execution of refinements later on, after we have introduced the refinement for the bitonic sort. Even though the refinement for the bitonic sort can be derived according to rigorous mathematical rules, we shall here simply state it informally.

Our refinement of the bitonic sort is:

```
            bitonic-sort(n):            sort(0,1,n+1)

            sort(base,step,0):          skip
            sort(base,step,1):          skip
{leng>1}    sort(base,step,leng):       sort(base,step*2,⌈leng/2⌉);
                                        sort(base+step,step*2,⌊leng/2⌋);
                                        segment(base,step,⌊leng/2⌋)
```

```
              segment(base,step,0):          skip
{leng>0}      segment(base,step,leng):       cs(base,base+step);
                                             segment(base+step*2,step,leng-1)
```

Refinement `sort` performs the bitonic sort as described in Sect. 1. Refinement `segment` performs the step of comparisons and interchanges by applying the appropriate comparator modules. Both refinements are qualified by three parameters, `base`, `step`, and `leng`, that identify a subsequence of `a`: `base` is the index of the first element, `step` is the difference of the indices of any two adjacent elements, and `leng` is the number of elements in the subsequence.

We can easily obtain a sequential execution of refinement `bitonic-sort(n)` by replacing all occurrences of the composition operator ';' by the sequential execution operator '→'. For instance, for an 8-element sequence (n=7), we obtain the following sequential trace:

```
tau(7)  =  cs(0,4)→cs(2,6)→cs(0,2)→cs(4,6)
                             →cs(1,5)→cs(3,7)→cs(1,3)→cs(5,7)
                                          →cs(0,1)→cs(2,3)→cs(4,5)→cs(6,7)
```

This execution is best explained by representing the problem of bitonic sorting as a tree.

Let us construct a binary tree of bitonic sequences whose root is the entire sequence `a`, and whose left and right subtrees are recursively constructed by splitting the root into subsequences as prescribed by the bitonic sorting algorithm. We call this tree the *sequence tree* of `a`. The sequence tree of an 8-element sequence is:

$$\begin{array}{c}
\langle a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7\rangle \\
/ \qquad\qquad \backslash \\
\langle a_0,a_2,a_4,a_6\rangle \qquad \langle a_1,a_3,a_5,a_7\rangle \\
/ \qquad \backslash \qquad\qquad / \qquad \backslash \\
\langle a_0,a_4\rangle \quad \langle a_2,a_6\rangle \quad \langle a_1,a_5\rangle \quad \langle a_3,a_7\rangle \\
/ \ \backslash \qquad / \ \backslash \qquad / \ \backslash \qquad / \ \backslash \\
\langle a_0\rangle\ \langle a_4\rangle \quad \langle a_2\rangle\ \langle a_6\rangle\ \langle a_1\rangle\ \langle a_5\rangle \quad \langle a_3\rangle\ \langle a_7\rangle
\end{array}$$

At each node $\langle a_{i_1},a_{i_2},a_{i_3},a_{i_4},\ldots\rangle$, the bitonic sorting algorithm requires an application of comparator modules $\mathtt{cs}(i_1,i_2);\mathtt{cs}(i_3,i_4);\ldots$, which we call a *segment*. The following *segment tree* corresponds to the previous sequence tree:

```
                cs(0,1);cs(2,3);cs(4,5);cs(6,7)
                         /             \
           cs(0,2);cs(4,6)          cs(1,3);cs(5,7)
            /          \              /         \
      cs(0,4)       cs(2,6)      cs(1,5)      cs(3,7)
```

Segments of leaves in the sequence tree are null and are not represented in the segment tree.

We can now view the sequential trace `tau(7)` of the bitonic sort as the post-order traversal of segments in the segment tree. `tau(7)` has length 12. In general, $\mathtt{tau}(2^k-1)$ has length $2^{k-1}k$. The refinement works for all bitonic sequences, but we choose to consider only sequences whose length `n+1` is a power k of 2. Such sequences yield complete sequence and segment trees. Also, for such sequences, the bitonic sort can be extended to a network which does not require the bitonic order of its input, as mentioned in Sect. 1. With $n=2^k-1$, trace tau(n) has a length of order O(n log n).

To speed up the execution of refinement bitonic-sort(n), we have to declare some program properties that permit us to relax the sequencing prescribed by trace tau. The crucial property for the compression of traces by concurrency is *independence*. For instance, program components that do not share any variables are independent. We can declare the independence of two program components S1 and S2 by writing S1||S2. If S1 and S2 are independent, their composition S1;S2 may be implemented by execution in parallel. We write this as <S1 S2> and call it a *parallel command*.[3] Of course, execution of independent S1 and S2 in sequence is also permitted, in fact, in either order: S1→S2 or S2→S1. Commuting the execution order of independent program parts in a trace may lead to a higher degree of concurrency than immediately merging them into a parallel command. See [5, 6] for more about independence.

Two comparator modules $cs(i_1,i_2)$ and $cs(j_1,j_2)$ are *disjoint* if they do not overlap, i.e., if $i_1 \neq j_1$, $i_1 \neq j_2$, $i_2 \neq j_1$, and $i_2 \neq j_2$. Since disjoint comparator modules do not share any variables, they may be declared independent:

$$i_1 \neq j_1 \wedge i_1 \neq j_2 \wedge i_2 \neq j_1 \wedge i_2 \neq j_2 \quad \Rightarrow \quad cs(i_1,i_2) \,||\, cs(j_1,j_2)$$

On the basis of this independence declaration, we may transform trace tau to obtain concurrency. Observe that any two distinct segments x and y in the segment tree which are not in an ascendant/descendant relationship have no common elements. Such x and y are independent, and we can commute them or make them parallel. For instance, we can commute all segments that are on the same level in the tree (i.e., that have the same distance from the root) into adjacency:

```
tau'(7) = cs(0,4)→cs(2,6)→cs(1,5)→cs(3,7)
                              →cs(0,2)→cs(4,6)→cs(1,3)→cs(5,7)
                                                  →cs(0,1)→cs(2,3)→cs(4,5)→cs(6,7)
```

Then we can merge each level into one parallel command:

```
tau~(7) = <cs(0,4) cs(2,6) cs(1,5) cs(3,7)>
                              →<cs(0,2) cs(4,6) cs(1,3) cs(5,7)>
                                                  →<cs(0,1) cs(2,3) cs(4,5) cs(6,7)>
```

The parallel trace tau~(7) is of length 3, with a concurrency degree of 4. In general, $tau^{\sim}(2^k-1)$ is of length k, with a concurrency degree of 2^{k-1}. With $n=2^k-1$, trace tau~(n) has a length of order O(log n) and a concurrency degree of order O(n).

[3]We abbreviate <S1 <S2 <... Sn>>> to <S1 S2 ... Sn>.

3. The Automated Proof of Trace Transformations

We have implemented our theory of trace transformations in Boyer & Moore's mechanized logic [1]. Boyer & Moore express terms of first-order predicate logic in a LISP-like functional form.[4] Predicates are functions with a boolean range. Functions can be *declared* (submitted without a function body) or *defined* (submitted with a function body), and facts can be asserted (submitted as an *axiom*) or proved (submitted as a *lemma*). There are no quantifiers. A variable that appears free in a term is taken as universally quantified. For example, the term

$(\mathtt{NUMBERP\ X}) \Rightarrow \mathtt{X} < \mathtt{X+1}$

expresses the fact that any number is smaller than the same number incremented by `1`. `NUMBERP` recognizes numbers. Two basic types of inductively constructed objects in the logic are relevant to our application: the natural number and the ordered pair. They closely resemble the number and ordered pair of LISP. The ordered pair `(CONS t1 t2)` of the two terms `t1` and `t2` may be abbreviated `(t1 . t2)` and lists can be formed by nesting ordered pairs, as in LISP. E.g., list `(t1 t2 ... tn)` of the `n` terms `t1,...,tn` is really `(t1 . (t2 . (... (tn . NIL))))`. Other object types may be added by the user. For instance, we add the object type $(\mathtt{PAIR}\ \mathtt{i}_1\ \mathtt{i}_2)$ or, abbreviated, $(\mathtt{i}_1\mathtt{:i}_2)$ which is a second kind of ordered pair - its components must be numbers. We use this object type to represent comparator modules.

The theorem proving program employs a number of heuristics in the attempt to establish the validity of a conjecture. Simplification (i.e., rewriting into a simpler or "normal" form) and induction are the heuristics used most in the proof of the transformation of the bitonic sort. There are several ways to make use of a previously established lemma in subsequent proofs. Our proof employs previously proved lemmas as rewrite rules. Appropriately chosen lemmas, provided as rewrite rules, will steer the prover into the intended direction of the proof. If all other heuristics fail, the prover appeals to induction. The induction scheme is derived from an analysis of the recursive function definitions and the inductively constructed types involved in the conjecture.

This section sketches the implementation of the semantic theory that is necessary to prove the semantic equivalence of trace `tau` and trace $\mathtt{tau}^{\sim}$ of the previous section. App. A contains proof outlines of the central theorems. The complete history of the proof is documented in App. B. We suggest that readers without experience in Boyer & Moore's logic consult [1] before studying the appendices.

[4]For clarity, we shall here, unlike LISP and Boyer & Moore, keep some basic logic and arithmetic operations in infix notation.

3.1. Trace Semantics

Our goal is to prove the semantic equivalence of traces tau(n) and tau⁻(n). We represent a trace by a multi-level list. Alternate levels indicate sequential execution and parallel execution, in turn. For instance, if the top level of the list indicates sequential execution, then the second level indicates parallel execution, the third level indicates again sequential execution, etc. In the realm of sorting networks, we can represent traces as multi-level lists of pairs of numbers, where the top level represents sequential execution. For example, the sequential trace

```
tau(7) = cs(0,4)→cs(2,6)→cs(0,2)→cs(4,6)
                       →cs(1,5)→cs(3,7)→cs(1,3)→cs(5,7)
                                       →cs(0,1)→cs(2,3)→cs(4,5)→cs(6,7)
```

is represented as

```
(TAU 7) = '( (0:4) (2:6) (0:2) (4:6)
                         (1:5) (3:7) (1:3) (5:7)
                                     (0:1) (2:3) (4:5) (6:7) )
```

and the parallel trace

```
tau⁻(7) = <cs(0,4) cs(2,6) cs(1,5) cs(3,7)>
                        →<cs(0,2) cs(4,6) cs(1,3) cs(5,7)>
                                         →<cs(0,1) cs(2,3) cs(4,5) cs(6,7)>
```

is represented as

```
(TAU⁻ 7) = '( ((0:4) (2:6) (1:5) (3:7))
                         ((0:2) (4:6) (1:3) (5:7))
                                      ((0:1) (2:3) (4:5) (6:7)) )
```

where $(i_1:i_2)$ denotes our new object type that represents a comparator module $cs(i_1,i_2)$.

We give traces weakest precondition semantics [7]. The weakest precondition of a fixed program S is a function $wp_S(R)$ that takes a postcondition R and maps it on the weakest possible constraints under which program S terminates and establishes R [3]. To give a programming language weakest precondition semantics, one must provide weakest preconditions for the smallest possible programs in the language and for combining smaller into bigger programs. The smallest possible sorting networks are the null statement and the comparator module. We need not implement the weakest precondition of null, because traces do not contain nulls (null has the empty trace). We declare the weakest precondition $wp_{cs(i_1,i_2)}(R)$ of comparator module $cs(i_1,i_2)$ as a function

Declaration

```
(CS I R)
```

where I denotes a pair, $(i_1:i_2)$, and R denotes a postcondition. Since we are only interested in the equality of weakest preconditions and not in their actual values, we need only *declare*, not *define* function CS. We need to know very little about the weakest precondition of the comparator module for the purpose of trace transformations and choose to add this information as axioms rather than inferring it from a full-fledged definition of function CS. We add two axioms. One restricts the domain of comparator modules to pairs of numbers:

Axiom CS-TAKES-PAIRS:

```
    (NOT (PAIRP I)) ⇒ ((CS I R) = F)
```

Axiom CS-TAKES-PAIRS states that the precondition of CS for any non-pair and postcondition is false,[5] i.e., that such a CS is not permitted. PAIRP recognizes pairs. The other axiom expresses the "rule of the excluded miracle" (Dijkstra's first healthiness criterion [3]) for comparator modules:

Axiom CS-IS-NOT-MIRACLE:

```
    (CS I F) = F
```

Axiom CS-IS-NOT-MIRACLE states that the precondition of any CS with false postcondition is false, i.e., a comparator module cannot establish "false".

Our way of combining smaller into bigger traces is by composition (i.e., execution in sequence or in parallel). To determine the weakest precondition of some trace L that is composed of comparator modules CS for postcondition R, we define a function M-CS. As for subsequent defined functions that we introduce, we shall first present the definition of M-CS and then explain its function body:

Definition

```
    (M-CS FLAG L R)
       =
    (IF (NOT (LISTP L))
        (IF L=NIL
            R
            (CS L R))
        (IF FLAG='PAR
            (IF (ARE-IND-CS (ALL-ATOMS (CAR L))
                            (ALL-ATOMS (CDR L)))
                (M-CS 'SEQ (CAR L) (M-CS 'PAR (CDR L) R))
                F)
            (M-CS 'PAR (CAR L) (M-CS 'SEQ (CDR L) R)))))
```

M-CS composes calls to CS as prescribed by trace L. Beside L and R, M-CS takes a FLAG that signals whether the trace is to be executed in sequence (FLAG='SEQ) or in parallel (FLAG='PAR). In accordance with our trace representation, FLAG='SEQ in top-level calls and FLAG alternates with every recursive call.

When FLAG='PAR, the trace represents a parallel command and its elements must be checked for independence. Like weakest preconditions, independence properties must be provided for the smallest program parts that are affected by a trace transformation and for combinations of these program parts. The smallest program parts affected by trace transformations of sorting networks are comparator modules. Just as we do not provide the complete weakest precondition semantics of comparator modules,

[5] In Boyer & Moore's logic, F stands for "false" and T stands for "true".

we do not provide a complete characterization of the independence of comparator modules but express it, again, by a declared function

Declaration

```
    (IND-CS I J)
```

where I and J are pairs which represent comparator modules. As we did with CS, we characterize IND-CS by axiom. For instance, we establish that IND-CS is a predicate:

Axiom IND-CS-IS-PREDICATE:

```
    (OR (TRUEP (IND-CS I J)) (FALSEP (IND-CS I J)))
```

In the following section on trace transformations, we shall discuss what other properties of function IND-CS we need to know.

We may now determine the independence of traces of comparator modules with defined functions that employ IND-CS appropriately. We define three functions.

IS-IND-CS establishes the mutual independence of one comparator module I with a trace L of comparator modules:

Definition

```
    (IS-IND-CS I L)
       =
    (IF (NOT (LISTP L))
        (IF L=NIL
            T
            (IND-CS I L))
        (AND (IND-CS I (CAR L))
             (IS-IND-CS I (CDR L))))
```

ARE-IND-CS establishes the mutual independence of all comparator modules in a trace L1 with all comparator modules in a trace L2:

Definition

```
    (ARE-IND-CS L1 L2)
       =
    (IF (NOT (LISTP L1))
        (IF L=NIL
            T
            (IS-IND-CS L1 L2))
        (AND (IS-IND-CS (CAR L1) L2)
             (ARE-IND-CS (CDR L1) L2)))
```

If the two members of a parallel command pass test ARE-IND-CS, function M-CS gives their parallel execution the semantics of their sequential execution.

A third function, TOTALLY-IND-CS, determines the independence of all comparator modules of a trace L:

Definition

```
(TOTALLY-IND-CS)
   =
(IF (NOT (LISTP L))
    T
    (AND (IS-IND-CS (CAR L) (CDR L))
         (TOTALLY-IND-CS (CDR L)))))
```

If trace L passes test TOTALLY-IND-CS, the execution of all members of L has identical semantics in parallel as in sequence.

Note that IS-IND-CS, ARE-IND-CS, and TOTALLY-IND-CS are only interested in the comparator modules of traces, not in the traces' structure. Therefore, these functions expect traces in a "flattened" form, i.e., as single-level lists with all comparator modules in the trace enumerated from left to right. The flattening is performed by function ALL-ATOMS:

Definition

```
(ALL-ATOMS L)
   =
(IF (NOT (LISTP L))
    (IF L=NIL
        NIL
        (LIST L))
    (APPEND (ALL-ATOMS (CAR L)) (ALL-ATOMS (CDR L))))
```

APPEND appends two lists. It differs from the regular LISP (and Boyer & Moore) append function which only works for proper lists, i.e., lists that end with NIL. Our APPEND works for all lists.

This concludes our implementation of the trace semantics. The semantic equivalence of tau~ and tau can now be formally expressed by the following equality:

```
(M-CS 'SEQ (TAU~ N) R) = (M-CS 'SEQ (TAU N) R)
```

In later proof outlines, we shall denote semantic equivalence by a special symbol: '$\equiv$'. For example, we would write the previous formula as:

```
(TAU~ N) ≡ (TAU N)
```

3.2. Trace Transformation

We are now at the point where we can begin formulating the transformation of tau into tau~ in Boyer & Moore's logic. Remember that the transformation rests on our independence declaration of comparator modules in Sect. 2. We exploit this declaration in two steps: one of commutations and a second of parallel merges. The step of merges is based on the following theorem:

Lemma M-CS-TOTALLY-IND:

```
(TOTALLY-IND-CS (ALL-ATOMS L))
   ⇒  ( (M-CS 'PAR L R) = (M-CS 'SEQ L R) )
```

It is proved by an induction scheme that mirrors the recursive definition of function M-CS. More general theorems that correspond to transformation rules (G3i) and (G3ii) in our original formalism (Sect. 5.2 of [7]) are also part of the implemented theory, but we do not use them in the transformation of the bitonic sort.

To express commutations, we must be more specific about the meaning of "independence". The declaration of IND-CS does not provide any clues. Just as about declared function CS, we need not know much about IND-CS for the purpose of trace transformations. For one, we must be able to conclude that independent comparator modules may be commuted:

Axiom IND-CS-IMPLIES-COMMUTATIVITY:

```
(IND-CS I J)  ⇒  ( (CS J (CS I R)) = (CS I (CS J R)) )
```

If we instantiate both FLAG1 and FLAG2 to 'SEQ, the following theorem enables commutations:

Lemma ARE-IND-CS-IMPLIES-COMMUTATIVITY:

```
(ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
   ⇒  (    (M-CS FLAG1 L1 (M-CS FLAG2 L2 R))
         = (M-CS FLAG2 L2 (M-CS FLAG1 L1 R)) )
```

Its proof is, again, based on an induction suggested by the definition of M-CS.

Just as we cannot compute weakest preconditions of comparator modules with declared function CS, we cannot determine their independence with declared function IND-CS. However, while we are not interested in the actual weakest preconditions of comparator modules, we do need to know about the circumstances of their independence. We established in Sect. 2 that two comparator modules are independent if they do not access common sequence elements, i.e., if they do not overlap. Our final axiom about IND-CS expresses this fact:

Axiom NO-OVERLAP-IND-CS:

```
(NO-OVERLAP I J)  ⇒  (IND-CS I J)
```

Function NO-OVERLAP identifies non-overlap:

Definition

```
(NO-OVERLAP I J)
   =
(AND (PAIRP I)
     (PAIRP J)
     (FIRST I)≠(FIRST J)
     (FIRST I)≠(SECOND J)
     (SECOND I)≠(FIRST J)
     (SECOND I)≠(SECOND J))
```

FIRST and SECOND access a pair's components: (FIRST $(i_1:i_2)$)=i_1 and (SECOND $(i_1:i_2)$)=i_2. Additional defined functions HAS-NO-OVERLAP, HAVE-NO-OVERLAP, and TOTALLY-NO-OVERLAP are exactly identical to IS-IND-CS, ARE-IND-CS, and TOTALLY-IND-CS, respectively, with calls to

NO-OVERLAP in place of calls to IND-CS. Theorems stating that each of the three overlap functions implies its respective independence counterpart can be proved from axiom NO-OVERLAP-IND-CS.

This concludes the implementation of our *basic theory*: the part that applies to *all* transformations of sorting networks. One might call this our metatheory of sorting networks, since it deals with properties of the programming language per se, not with properties of one specific sorting network. The next section describes our application in that theory: the transformation of the bitonic sort.

3.3. Transformation of the Bitonic Sort

Sect. 3.1 introduces informally the representation of traces in Boyer & Moore's logic. We shall now formally define the traces relevant to the transformation of the bitonic sort. To derive the parallel trace tau~ from the sequential trace tau, we formulated an intermediate trace tau'. Trace tau' is derived from tau by a step of commutations, and tau~ is derived from tau' by a step of parallel merges. We shall describe the implementation of tau first, then of tau~, and then of tau'.

Trace tau is defined by the following three functions:

Definition

```
(TAU N)
   =
(SORT 0 1 N+1)
```

Definition

```
(SORT BASE STEP LENG)
   =
(IF (OR LENG=0 LENG=1)
    NIL
    (APPEND (SORT BASE STEP+STEP LENG/2)
            (APPEND (SORT BASE+STEP STEP+STEP LENG/2)
                    (SEGMENT BASE STEP LENG/2))))
```

Definition

```
(SEGMENT BASE STEP LENG)
   =
(IF LENG=0
    NIL
    (CONS (PAIR BASE BASE+STEP)
          (SEGMENT BASE+STEP+STEP STEP LENG-1)))
```

Functions TAU, SORT, and SEGMENT correspond exactly to refinements bitonic-sort, sort, and segment of Sect. 2, except that refinement composition is replaced by trace composition (sequential execution, to be more precise). Parameters N, BASE, STEP, and LENG have the same meaning as their counterparts in the refinements.

The refinement uses real division, but the mechanized logic provides only integer division: the function QUOTIENT [1] which we denote here also by the infix '/'. Remember that we decided to consider only sequences whose length is a power of 2. With this restriction, integer division suffices in the trace definitions.

Trace tau~ is defined by the three functions:

Definition

```
(TAU~ N)
   =
(PAR-CMDS '(0) 1 N+1)
```

Definition

```
(PAR-CMDS SEQ-HEAD STEP LENG)
   =
(IF (OR LENG=0 LENG=1)
    NIL
    (APPEND (PAR-CMDS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
            (LIST (SAME-LEVEL SEQ-HEAD STEP LENG))))
```

Definition

```
(SAME-LEVEL SEQ-HEAD STEP LENG)
   =
(IF (NOT (LISTP SEQ-HEAD))
    NIL
    (APPEND (SEGMENT (CAR SEQ-HEAD) STEP LENG/2)
            (SAME-LEVEL (CDR SEQ-HEAD) STEP LENG)))
```

The parallel trace is not derived from a refinement. We have to explain it independently. Function TAU~ uses function PAR-CMDS to compose parallel commands as described in Sect. 2. Each parallel command of TAU~ contains all the comparator modules at a fixed level of the segment tree. The comparator modules at a fixed level of the segment tree are collected by function SAME-LEVEL. Remember that every segment tree has a corresponding sequence tree. PAR-CMDS and SAME-LEVEL have three arguments that, together, describe the subsequences at a fixed level of the sequence tree: SEQ-HEAD, STEP, and LENG. SEQ-HEAD is the "sequence head", the list of the first elements of the subsequences at that level, STEP is the difference of any two adjacent elements of subsequences at that level, and LENG is the length of subsequences at that level. For the recursive definition of PAR-CMDS, we need to determine the SEQ-HEAD of the next lower level. For this purpose we define the following function:

Definition

```
(GEN-SEQ-HEAD SEQ-HEAD STEP)
   =
(IF (NOT (LISTP SEQ-HEAD))
    NIL
    (CONS (CAR SEQ-HEAD)
          (CONS (CAR SEQ-HEAD)+STEP
                (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP))))
```

Given SEQ-HEAD and STEP at some fixed level, function GEN-SEQ-HEAD computes SEQ-HEAD of the next lower level. For example, SEQ-HEAD of the 8-element sequence tree at the second level (the level below the root) is '(0 1), and STEP is 2. Therefore

```
(GEN-SEQ-HEAD '(0 1) 2) = '(0 2 1 3)
```

Trace tau' is defined as follows:

Definition

```
(TAU' N)
   =
(APPEND-LEVELS '(0) 1 N+1)
```

Definition

```
(APPEND-LEVELS SEQ-HEAD STEP LENG)
   =
(IF (OR LENG=0 LENG=1)
     NIL
     (APPEND (APPEND-LEVELS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
             (SAME-LEVEL SEQ-HEAD STEP LENG)))
```

TAU' uses function APPEND-LEVELS to compose the levels of the segment tree in bottom-up sequence. TAU' differs from TAU~ only by the lack of additional "parallel command" parentheses around each level.

With the definition of traces TAU, TAU~, and TAU', we are able to state the trace transformation of the bitonic sort. Our main theorem is:

Lemma TAU~-EQ-TAU:

```
(M-CS 'SEQ (TAU~ N) R) = (M-CS 'SEQ (TAU N) R)
```

Its proof applies one step of commutations:

Lemma TAU'-EQ-TAU:

```
(M-CS 'SEQ (TAU' N) R) = (M-CS 'SEQ (TAU N) R)
```

and one step of parallel merges:

Lemma TAU~-EQ-TAU':

```
(M-CS 'SEQ (TAU~ N) R) = (M-CS 'SEQ (TAU' N) R)
```

We shall now explain the proof of lemmas TAU'-EQ-TAU and TAU~-EQ-TAU' in more detail.

Boyer & Moore make the point that it is sometimes easier to prove a more general theorem than that in which one is interested [1]. Sometimes, a more general theorem may allow a more powerful or more convenient induction hypothesis.[6] We found it useful to generalize our trace transformation in the interest of a simpler mechanical proof.

[6]Boyer & Moore actually built the generalization of theorems as a heuristic into their prover.

In the definition of traces TAU, TAU~, and TAU', the arguments of SORT, PAR-CMDS, and APPEND-LEVELS describe the entire sequence, i.e., the root of the sequence tree, and theorems TAU'-EQ-TAU and TAU~-EQ-TAU' state the transformation of the entire trace, i.e., deal with the entire sequence tree. We shall generalize the trace transformation theorems to deal with traces corresponding to *any set* of sequence *subtrees* whose roots reside at some fixed level in the sequence tree. For a set of such roots specified by SEQ-HEAD, STEP, and LENG, the following function defines the appropriate generalized sequential trace:

Definition

```
(SUBTREES SEQ-HEAD STEP LENG)
   =
(IF (NOT (LISTP SEQ-HEAD))
    NIL
    (APPEND (SORT (CAR SEQ-HEAD) STEP LENG)
            (SUBTREES (CDR SEQ-HEAD) STEP LENG)))
```

Every call of function SORT represents a sequential trace that traverses a segment subtree in post-order. The root of this segment subtree is a segment corresponding to the subsequence represented by some element of SEQ-HEAD, and STEP and LENG. As an example, let us consider the third level (from the top) of the sequence tree of a 16-element sequence $\langle a_0, a_1, \ldots, a_{15}\rangle$. Its subsequences are $\langle a_0, a_4, a_8, a_{12}\rangle$, $\langle a_2, a_6, a_{10}, a_{14}\rangle$, $\langle a_1, a_5, a_9, a_{13}\rangle$, and $\langle a_3, a_7, a_{11}, a_{15}\rangle$. They are described by the following parameters: SEQ-HEAD='(0 2 1 3), STEP=4, and LENG=4. The segment subtrees whose roots are the segments corresponding to these subsequences are:

```
  (0:4) (8:12)         (2:6) (10:14)         (1:5) (9:13)          (3:7) (11:15)
    /      \             /       \             /       \              /       \
(0:8)      (4:12)    (2:10)      (6:14)    (1:9)       (5:13)     (3:11)      (7:15)
```

Thus, each call of SORT in (SUBTREES '(0 1 2 3) 4 4) traverses one of the previous segment subtrees in post-order:

```
(SUBTREES '(0 2 1 3) 4 4) = '( (0:8)  (4:12) (0:4) (8:12)
                               (2:10) (6:14) (2:6) (10:14)
                               (1:9)  (5:13) (1:5) (9:13)
                               (3:11) (7:15) (3:7) (11:15) )
```

Function SUBTREES is a generalization of our original trace TAU. SUBTREES with arguments that represent the set of only the entire sequence (i.e., the root of the entire sequence tree) equals TAU:

```
(SUBTREES '(0) 1 N+1) = (TAU N)
```

Before we can discuss the generalized trace transformation theorems for the bitonic sort, we have to add a "recognizer" of legal sets of subsequences. Remember that the set of subsequences expected by SUBTREES must reside at some fixed level. The following function provides a sufficient condition that the subsequences represented by SEQ-HEAD, STEP, and LENG reside at the same level and have no duplication:

Definition

```
(SEQ-HEADP SEQ-HEAD STEP)
   =
(IF (NOT (LISTP SEQ-HEAD))
    T
    (AND STEP≠0
         (NUMBERP (CAR SEQ-HEAD))
         (CAR SEQ-HEAD)<STEP
         (NOT (MEMBER (CAR SEQ-HEAD) (CDR SEQ-HEAD)))
         (SEQ-HEADP (CDR SEQ-HEAD) STEP)))
```

Function SEQ-HEADP determines that all elements of SEQ-HEAD are distinct natural numbers and less than STEP. For example, for the subsequences at the third level of the sequence tree of a 16-element sequence:

```
(SEQ-HEADP '(0 2 1 3) 4) = T
```

If SEQ-HEAD and STEP at some fixed level satisfy property SEQ-HEADP, the following theorem ensures that the sequence head at the next lower level, generated by GEN-SEQ-HEAD, satisfies SEQ-HEADP with doubled step, as appropriate for that level:

Lemma GEN-SEQ-HEAD-IS-SEQ-HEADP:

```
(SEQ-HEADP SEQ-HEAD STEP)
   ⇒  (SEQ-HEADP (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP)
```

The proof proceeds by an induction suggested by function SEQ-HEADP, but requires the previous proof of some technicalities (see App. A, lemma GEN-SEQ-HEAD-IS-NOT-MEMBER).

Now we are ready to explain the generalized trace transformation theorems. The following theorem is the generalized version of the step of commutations, TAU'-EQ-TAU:

Lemma APPEND-LEVELS-EQ-SUBTREES:

```
(SEQ-HEADP SEQ-HEAD STEP)
   ⇒  (   (M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG) R)
        = (M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R) )
```

This theorem is proved by commutation of all comparator modules at a fixed level into adjacency, as we explained in Sect. 2. We cannot expect the prover to discover this transformation strategy without help, but must communicate it with the following auxiliary lemma:

Lemma SUBTREES-COMMUTATIVITY:

```
(SEQ-HEADP SEQ-HEAD STEP)
   ⇒  ( (M-CS 'SEQ
              (SUBTREES (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
              (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
             =
         (M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R) )
```

Let us illustrate by example of the previous subtrees of the 16-element sequence how the mechanical proof of lemma SUBTREES-COMMUTATIVITY proceeds. To satisfy the premise of the transformation, let us accept that SEQ-HEAD='(0 2 1 3), STEP=4, and LENG=4 represent a set of subsequences at some fixed level, the third level, of the 16-element sequence tree, i.e., let us assert:

```
(SEQ-HEADP '(0 2 1 3) 4) = T
```

The proof of SUBTREES-COMMUTATIVITY is based on the induction suggested by the recursive definition of function GEN-SEQ-HEAD. The induction hypothesis is:[7]

```
(APPEND (SUBTREES '(2 6 1 5 3 7) 8 2) (SAME-LEVEL '(2 1 3) 4 4))
   ≡
(SUBTREES '(2 1 3) 4 4)
```

Here is an illustration of the induction step:

left-hand side:

```
(APPEND (SUBTREES (GEN-SEQ-HEAD '(0 2 1 3) 4) 8 2)
        (SAME-LEVEL '(0 2 1 3) 4 4))
```

≡ {open GEN-SEQ-HEAD}

```
(APPEND (SUBTREES '(0 4 2 6 1 5 3 7) 8 2) (SAME-LEVEL '(0 2 1 3) 4 4))
```

≡ {open SUBTREES and SAME-LEVEL, and apply M-CS-APPEND}

```
(APPEND (SORT 0 8 2)
        (APPEND (SORT 4 8 2)
                (APPEND (SUBTREES '(2 6 1 5 3 7) 8 2)
                        (APPEND (SEGMENT 0 4 2)
                                (SAME-LEVEL '(2 1 3) 4 4)))))
```

≡ {commutation: apply SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP
and ARE-IND-CS-IMPLIES-COMMUTATIVITY}

```
(APPEND (SORT 0 8 2)
        (APPEND (SORT 4 8 2)
                (APPEND (SEGMENT 0 4 2)
                        (APPEND (SUBTREES '(2 6 1 5 3 7) 8 2)
                                (SAME-LEVEL '(2 1 3) 4 4)))))
```

≡ {induction hypothesis}

```
(APPEND (SORT 0 8 2)
        (APPEND (SORT 4 8 2)
                (APPEND (SEGMENT 0 4 2)
                        (SUBTREES '(2 1 3) 4 4))))
```

right-hand side:

```
(SUBTREES '(0 2 1 3) 4 4)
```

≡ {open SUBTREES}

```
(APPEND (SORT 0 4 4) (SUBTREES '(2 1 3) 4 4))
```

≡ {open SORT, and apply M-CS-APPEND}

[7] Recall that we use symbol ≡ to denote semantic equivalence. Comments are phrased in curly brackets.

```
(APPEND (SORT 0 8 2)
        (APPEND (SORT 4 8 2)
                (APPEND (SEGMENT 0 4 2)
                        (SUBTREES '(2 1 3) 4 4))))
```

The proof uses commutation rule ARE-IND-CS-IMPLIES-COMMUTATIVITY of our basic theory (Sect. 3.2) to commute (SUBTREES '(2 6 1 5 3 7) 8 2) with (SEGMENT 0 4 2). In order to apply the commutation rule, we have to establish its hypothesis:

```
(HAVE-NO-OVERLAP (SUBTREES '(2 6 1 5 3 7) 8 2) (SEGMENT 0 4 2)) = T
```

We formulate, again, an auxiliary lemma to this effect:

Lemma SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP:

```
(SEQ-HEADP SEQ-HEAD STEP)
   ⇒ (HAVE-NO-OVERLAP (SUBTREES (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP)
                                STEP+STEP
                                LENG1)
                      (SEGMENT (CAR SEQ-HEAD) STEP LENG2))))
```

To prove it, we have to generalize again - this time for a purely technical reason. Boyer & Moore's logic requires a uniform substitution of all identical terms in a theorem. However, in the induction hypothesis of SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP, we need a substitution of (CDR SEQ-HEAD) for SEQ-HEAD in term (CDR SEQ-HEAD), but not in term (CAR SEQ-HEAD). We generalize by replacing the two terms (CAR SEQ-HEAD) and (CDR SEQ-HEAD) with variables, say, CAR-SEQ-HEAD and CDR-SEQ-HEAD, respectively.

The proof of the generalized version of SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP requires the non-overlap of certain trace parts. Establishing it has been the biggest challenge of the whole proof. We had to provide the prover with 25 lemmas of natural arithmetic, mostly of integer division.

Let us return now to theorem APPEND-LEVELS-EQ-SUBTREES. For its proof, we have to supply an induction hint that forces the proper substitution in the induction hypothesis. Boyer & Moore's prover permits us to suggest an induction scheme by a defined function of our choice. The following function models the induction scheme that we desire:

Definition

```
(INDUCTION-SCHEME SEQ-HEAD STEP LENG R)
   =
(IF (OR LENG=0 LENG=1)
    NIL
    (INDUCTION-SCHEME (GEN-SEQ-HEAD SEQ-HEAD STEP)
                      STEP+STEP
                      LENG/2
                      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R)))
```

Let us illustrate the proof of theorem APPEND-LEVELS-EQ-SUBTREES, again, using our four sequence subtrees of the 16-element bitonic sequence. The induction hypothesis is:

```
(APPEND-LEVELS '(0 4 2 6 1 5 3 7) 8 2)
   ≡
(SUBTREES '(0 4 2 6 1 5 3 7) 8 2)
```

Here is an outline of the induction step:

```
    (APPEND-LEVELS '(0 2 1 3) 4 4)
  ≡ {open APPEND-LEVELS and GEN-SEQ-HEAD, and apply M-CS-APPEND}
    (APPEND (APPEND-LEVELS '(0 4 2 6 1 5 3 7) 8 2) (SAME-LEVEL '(0 2 1 3) 4 4))
  ≡ {induction hypothesis}
    (APPEND (SUBTREES '(0 4 2 6 1 5 3 7) 8 2) (SAME-LEVEL '(0 2 1 3) 4 4))
  ≡ {apply SUBTREES-COMMUTATIVITY}
    (SUBTREES '(0 2 1 3) 4 4)
```

The commutations in this proof appeal to lemma SUBTREES-COMMUTATIVITY.

This concludes our discussion of the transformation step of commutations: theorem APPEND-LEVELS-EQ-SUBTREES. We still have to describe the transformation step of parallel merges.

The generalization of the step of parallel merges, TAU~-EQ-TAU', is:

Lemma PAR-CMDS-MERGE:

```
(SEQ-HEADP SEQ-HEAD STEP)
   ⇒  (    (M-CS 'SEQ (PAR-CMDS SEQ-HEAD STEP LENG) R)
         = (M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG) R) )
```

The proof is, again, based on induction hint INDUCTION-SCHEME. Let us illustrate the proof of theorem PAR-CMDS-MERGE with our, by now, familiar example. The induction hypothesis is:

```
(PAR-CMDS '(0 4 2 6 1 5 3 7) 8 2)
   ≡
(APPEND-LEVELS '(0 4 2 6 1 5 3 7) 8 2)
```

Here is an illustration of the induction step:

```
    (PAR-CMDS '(0 2 1 3) 4 4)
  ≡ {open PAR-CMDS and GEN-SEQ-HEAD, and apply M-CS-APPEND}
    (APPEND (PAR-CMDS '(0 4 2 6 1 5 3 7) 8 2) ((SAME-LEVEL '(0 2 1 3) 4 4)))
  ≡ {parallel merges: apply SAME-LEVEL-IS-TOTLALLY-NO-OVERLAP and M-CS-TOTALLY-IND}
    (APPEND (PAR-CMDS '(0 4 2 6 1 5 3 7) (SAME-LEVEL '(0 2 1 3) 4 4))
  ≡ {induction hypothesis}
    (APPEND (APPEND-LEVELS '(0 4 2 6 1 5 3 7) (SAME-LEVEL '(0 2 1 3) 4 4))
  ≡ {open APPEND-LEVELS, and apply M-CS-APPEND}
    (APPEND-LEVELS '(0 2 1 3) 4 4)
```

The proof uses independence rule M-CS-TOTALLY-IND of our basic theory (Sect. 3.2). To establish its prerequisites, we have to prove:

```
(TOTALLY-NO-OVERLAP (SAME-LEVEL '(0 2 1 3) 4 4)) = T
```

The proof, again, requires properties of integer division.

This concludes our discussion of the transformation step of parallel merges: theorem PAR-CMDS-MERGE.

With generalizations APPEND-LEVELS-EQ-SUBTREES and PAR-CMDS-MERGE at our disposal, the proofs of the more specific theorems TAU'-EQ-TAU and TAU~-EQ-TAU' are straight-forward. Only rewrites, no inductions are required. In fact, the proof of the main theorem, TAU~-EQ-TAU, would succeed without the previous proof of the specific theorems, only on the basis of the generalized theorems. We chose to force the prover to use the specific theorems (see App. B) to make the correspondence with the informal description of our transformation perfectly clear.

Our informal transformation (Sect. 2) was from sequential to parallel. In the mechanized proof, we rewrite in the reverse direction, from parallel to sequential, in order to avoid implementation problems. We need not be aware that the prover actually transforms TAU~ into TAU, not vice versa. The direction of transformation is of no consequence to the proof of the semantic equivalence.

We are at the end of the description of the mechanized proof. The proof rests on five axioms that are part of our basic theory. They reflect properties of comparator modules and their independence that we are willing to accept without certification. Relative to the basic theory, our application is completely defined and certified. See App. A for proof outlines of theorems SUBTREES-COMMUTATIVITY, APPEND-LEVELS-EQ-SUBTREES, and PAR-CMDS-MERGE. App. B contains the complete proof session.

4. Conclusions

Our interest is in the mechanical support of formal reasoning about properties of programming languages and programs. Presently, we focus on the transformation of program executions to derive concurrency. The bitonic sort is the third in a series of sorting networks for which we have mechanically certified trace transformations [8].

Our approach differs from most mechanical verification systems in that we make the formal semantic definition of the programming language itself available to the prover. The more popular approach is to employ some ad hoc device instead which stands between the program to be verified and the formulas to be proved, for instance, an informally derived verification condition generator. While the use of a verification condition generator permits highly automated and reasonably practical verification systems, the price paid is that the mechanical prover cannot be used for reasoning about program properties other than the ones handed to it by the verification condition generator. Making the formal semantics of the language available to the prover enables independent reasoning about program properties and metareasoning about properties of the language itself. (Here, the metareasoning is the more significant gain. For example, our metareasoning is about equivalences of program executions.) Also, one has to believe the correctness of only one computer program: the mechanical deduction system. One does not have to rely additionally on the correct implementation of a second program (the verification condition generator) and the fact that it corresponds to the formal semantic definition of the programming language. Understandably, this puts a much heavier burden on the deduction system.

However, while mechanical proofs of language properties on the basis of a formal semantics are far away from complete automation, our experience in mechanized proofs of trace equivalences suggests that a mechanized prover will, in general, follow the clean strategy of an on-paper proof, if it is communicated properly [8]. In the case of the bitonic sort, we communicated the structure of the proof by definition of three traces: the sequential trace `tau`, the intermediate trace `tau'`, and the final parallel trace `tau~`, and the formulation of two theorems: one equating `tau` and `tau'` by a commutation argument, and one equating `tau'` and `tau~` by a parallel merge argument. The hardest part of the whole proof was to establish certain properties of integer division, even though we did not dwell here on that particular aspect. We have concentrated our efforts on a nice implementation of the theory of trace transformations, not the theory of natural arithmetic.

Still, even though successful and with undeniable structure - the tediousity of the mechanical proof, compared to the informal description of the transformation, cannot be denied. Our point of view is that it should be expected. A mechanized proof does not permit any short-cuts. Each ever so little detail has to be formalized. It is the producer of the program or programming language about which is reasoned who has to suffer from this stringent requirement. The consumer reaps the benefits. Besides believing the correctness of the theorem proving program, he only has to be convinced that the theorem to be proved is appropriately represented in the mechanized logic. He does not have to be concerned with any aspects of the proof. In this example, the consumer must believe that `M-CS` properly defines the trace semantics, that `TAU` and `TAU~` properly define the sequential and parallel trace, and that no illegal axiomatic assumptions have been made. The manner in which the prover certifies theorem `TAU~-EQ-TAU` is of no concern to him.

Ultimately, the producer benefits as well: while the theorems about his product will be harder to establish, they will be easier to sell. It must be added that not every programming product justifies completely formal and mechanized scrutiny. There must be a substantial interest in the product's precise properties because the cost of the proof will be high.

Acknowledgement

We are grateful to the attendants of the Boyer-Moore Verification Seminar for critical comments that helped improve the presentation of this proof.

References

1. Boyer, R. S., and Moore, J S. *A Computational Logic.* Academic Press, 1979.

2. Boyer, R. S., and Moore, J S. A Theorem Prover for Recursive Functions, a User's Manual. Computer Science Laboratory, SRI International, 1979.

3. Dijkstra, E. W. *A Discipline of Programming.* Series in Automatic Computation, Prentice-Hall, 1976.

4. Knuth, D. E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973. Sect. 5.3.4.

5. Lengauer, C., and Hehner, E. C. R. "A Methodology for Programming with Concurrency: An Informal Presentation". *Science of Computer Programming 2,* 1 (Oct. 1982), 1-18.

6. Lengauer, C. "A Methodology for Programming with Concurrency: The Formalism". *Science of Computer Programming 2,* 1 (Oct. 1982), 19-52.

7. Lengauer, C. A Methodology for Programming with Concurrency. CSRG-142, Computer Systems Research Group, University of Toronto, Apr., 1982.

8. Lengauer, C., and Huang, C.-H. The Static Derivation of Concurrency and Its Mechanized Certification. Proc. NSF-SERC Seminar on Concurrency, Lecture Notes in Computer Science, Springer-Verlag, 1984. To appear.

Appendix A: Proof Outlines

Proof outlines display induction steps only. Comments are added in curly brackets.

<u>SUBTREES-COMMUTATIVITY:</u>

left-hand side:

```
(M-CS 'SEQ (SUBTREES (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {open GEN-SEQ-HEAD}

```
(M-CS 'SEQ (SUBTREES (CONS (CAR SEQ-HEAD)
                           (CONS (CAR SEQ-HEAD)+STEP
                                 (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP))))
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {open SUBTREES and SAME-LEVEL, and apply M-CS-APPEND}

```
(M-CS 'SEQ (SORT (CAR SEQ-HEAD) STEP+STEP LENG/2)
      (M-CS 'SEQ (SORT (CAR SEQ-HEAD)+STEP STEP+STEP LENG/2)
            (M-CS 'SEQ (SUBTREES (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP)
                                 STEP+STEP LENG/2)
                  (M-CS 'SEQ (SEGMENT (CAR SEQ-HEAD) STEP STEP+STEP LENG/2)
                        (M-CS 'SEQ (SAME-LEVEL (CDR SEQ-HEAD) STEP LENG) R)))))
```

= {apply SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP
and ARE-IND-CS-IMPLIES-COMMUTATIVITY}

```
(M-CS 'SEQ (SORT (CAR SEQ-HEAD) STEP+STEP LENG/2)
      (M-CS 'SEQ (SORT (CAR SEQ-HEAD)+STEP STEP+STEP LENG/2)
            (M-CS 'SEQ (SEGMENT (CAR SEQ-HEAD) STEP STEP+STEP LENG/2)
                  (M-CS 'SEQ (SUBTREES (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP)
                                       STEP+STEP LENG/2)
                        (M-CS 'SEQ (SAME-LEVEL (CDR SEQ-HEAD) STEP LENG) R)))))
```

= {induction hypothesis}

```
(M-CS 'SEQ (SORT (CAR SEQ-HEAD) STEP+STEP LENG/2)
      (M-CS 'SEQ (SORT (CAR SEQ-HEAD)+STEP STEP+STEP LENG/2)
            (M-CS 'SEQ (SEGMENT (CAR SEQ-HEAD) STEP STEP+STEP LENG/2)
                  (M-CS 'SEQ (SUBTREES (CDR SEQ-HEAD) STEP LENG) R))))
```

right-hand side:

```
(M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R)
```

= {open SUBTREES, and apply M-CS-APPEND}

```
(M-CS 'SEQ (SORT (CAR SEQ-HEAD) STEP LENG)
      (M-CS 'SEQ (SUBTREES (CDR SEQ-HEAD) STEP LENG) R))
```

= {open SORT, and apply M-CS-APPEND}

```
(M-CS 'SEQ (SORT (CAR SEQ-HEAD) STEP+STEP LENG/2)
      (M-CS 'SEQ (SORT (CAR SEQ-HEAD)+STEP STEP+STEP LENG/2)
            (M-CS 'SEQ (SEGMENT (CAR SEQ-HEAD) STEP STEP+STEP LENG/2)
                  (M-CS 'SEQ (SUBTREES (CDR SEQ-HEAD) STEP LENG) R))))
```

Q.E.D.

<u>APPEND-LEVELS-EQ-SUBTREES:</u>

```
(M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG))
```

= {open APPEND-LEVELS, and apply M-CS-APPEND}

```
(M-CS 'SEQ (APPEND-LEVELS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {induction hypothesis}

```
(M-CS 'SEQ (SUBTREES (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {apply SUBTREES-COMMUTATIVITY}

```
(M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R)
```

Q.E.D.

<u>PAR-CMDS-MERGE:</u>

```
(M-CS 'SEQ (PAR-CMDS SEQ-HEAD STEP LENG) R)
```

= {open PAR-CMDS, and apply M-CS-APPEND}

```
(M-CS 'SEQ (PAR-CMDS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (LIST (SAME-LEVEL SEQ-HEAD STEP LENG)) R))
```

= {open M-CS}

```
(M-CS 'SEQ (PAR-CMDS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'PAR (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {apply PAR-CMDS-IS-TOTALLY-NO-OVERLAP and M-CS-TOTALLY-IND}

```
(M-CS 'SEQ (PAR-CMDS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {induction hypothesis}

```
(M-CS 'SEQ (APPEND-LEVELS (GEN-SEQ-HEAD SEQ-HEAD STEP) STEP+STEP LENG/2)
      (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
```

= {open APPEND-LEVELS, and apply M-CS-APPEND}

```
(M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG R))
```

Q.E.D.

Appendix B: Events of Proof Session

This appendix presents all events in the order in which they have been accepted by the theorem prover. We use five kinds of events: declaration of a function, definition of a function, addition of a shell, addition of an axiom, and proof of a lemma. We shall briefly review the input command format of each. The User's Manual [2] explains how to run proof sessions, in general.

1. Function Declaration: `(DCL name args)`

   `DCL` declares `name` to be an undefined function with formal arguments `args`.

2. Function Definition: `(DEFN name args body hints)`

   `DEFN` defines a function name `name` with formal arguments `args` and with body `body`. Before admission of the function, the prover attempts to certify its termination by identifying a well-founded relation such that some measure of `args` gets smaller in every recursive call. In some cases, this relation and measure must be provided in the fourth argument `hints`.

3. Add Shell: `(ADD-SHELL const btm recog acces)`

   `ADD-SHELL` defines a new type of object. `const`, the type's *constructor* function, takes `n` arguments and returns an `n`-tuple object of the new type. `n` is the number of *accessor* functions that access components of objects of the type. Optional `btm` is the *bottom object* of the type, `recog` is the *recognizer* function that identifies an object of the type, and `acces` specifies all accessor functions.

4. Add Axiom: `(ADD-AXIOM name types term)`

   `ADD-AXIOM` adds a new axiom. The name of the axiom is `name`, `types` specifies the ways in which the axiom is used by the prover, and the statement of the axiom is `term`. All of our axioms are of type `REWRITE`, i.e., are used as rewrite rules.

5. Prove Lemma: `(PROVE-LEMMA name type term hints)`

   `PROVE-LEMMA` attempts to prove the conjecture `term` and remember it as a lemma named `name`. Only successfully proved lemmas are admitted as events. Lemma `name` will be used according to `types`. Our lemmas are all used as rewrite rules. The fourth argument `hints` may contain several kinds of directives to aid the proof. We use the following hints:

   `(INDUCT (name args))`

   Use the induction scheme reflected by the recursive definition of function `(name args)`.

   `(DISABLE` $\mathtt{ev}_1$ ... $\mathtt{ev}_n$`)`

   Prevent the use of events $\mathtt{ev}_1$ to a_n in the proof.

   `(USE` $\mathtt{lemma}_1$ ... $\mathtt{lemma}_n$`)`

   Enforce the use of axioms or lemmas $\mathtt{lemma}_1$ to $\mathtt{lemma}_n$. Each $\mathtt{lemma}_i$ has the form `(name (`v_1 t_1`) ... (`v_n t_n`))`, where `name` is the name of the axiom or the lemma to be used, v_i is one of the free variables of `name`, and t_i is a substitution term for v_i.

The following list of events contains:	2 declared functions,
	22 defined functions,
	1 shell,
	5 axioms and
	75 lemmas, i.e.,
	-----
	105 events.

BASIC THEORY OF SORTING NETWORKS

Trace Composition

```
(DEFN APPEND (X Y)
      (IF (NLISTP X)
          (IF (EQUAL X NIL)
              Y
              (CONS X Y))
          (CONS (CAR X) (APPEND (CDR X) Y))))


(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z) (APPEND X (APPEND Y Z))))


(DEFN ALL-ATOMS (L)
      (IF (NLISTP L)
          (IF (EQUAL L NIL)
              NIL
              (LIST L))
          (APPEND (ALL-ATOMS (CAR L)) (ALL-ATOMS (CDR L)))))


(PROVE-LEMMA ALL-ATOMS-APPEND (REWRITE)
             (EQUAL (ALL-ATOMS (APPEND X Y))
                    (APPEND (ALL-ATOMS X) (ALL-ATOMS Y))))
```

Trace Semantics

```
(DCL CS (I R))


(DCL IND-CS (I J))


(ADD-AXIOM IND-CS-IS-PREDICATE (REWRITE)
           (OR (TRUEP (IND-CS I J)) (FALSEP (IND-CS I J))))
```

```
(DEFN IS-IND-CS (I L)
      (IF (NLISTP L)
          (IF (EQUAL L NIL)
              T
              (IND-CS I L))
          (AND (IND-CS I (CAR L)) (IS-IND-CS I (CDR L)))))

(PROVE-LEMMA IS-IND-CS-APPEND (REWRITE)
             (EQUAL (IS-IND-CS I (APPEND L1 L2)) (AND (IS-IND-CS I L1) (IS-IND-CS I L2))))

(DEFN ARE-IND-CS (L1 L2)
      (IF (NLISTP L1)
          (IF (EQUAL L1 NIL)
              T
              (IS-IND-CS L1 L2))
          (AND (IS-IND-CS (CAR L1) L2) (ARE-IND-CS (CDR L1) L2))))

(PROVE-LEMMA ARE-IND-CS-APPEND-RIGHT (REWRITE)
             (EQUAL (ARE-IND-CS L1 (APPEND L2 L3)) (AND (ARE-IND-CS L1 L2) (ARE-IND-CS L1 L3))))

(PROVE-LEMMA ARE-IND-CS-APPEND-LEFT (REWRITE)
             (EQUAL (ARE-IND-CS (APPEND L1 L2) L3) (AND (ARE-IND-CS L1 L3) (ARE-IND-CS L2 L3))))

(DEFN TOTALLY-IND-CS (L)
      (IF (NLISTP L)
          T
          (AND (IS-IND-CS (CAR L) (CDR L)) (TOTALLY-IND-CS (CDR L)))))

(PROVE-LEMMA TOTALLY-IND-CS-APPEND (REWRITE)
             (IMPLIES (TOTALLY-IND-CS (APPEND L1 L2))
                      (AND (ARE-IND-CS L1 L2) (TOTALLY-IND-CS L1) (TOTALLY-IND-CS L2))))

(DEFN M-CS (FLAG L R)
      (IF (NLISTP L)
          (IF (EQUAL L NIL)
              R
              (CS L R))
          (IF (EQUAL FLAG 'PAR)
              (IF (ARE-IND-CS (ALL-ATOMS (CAR L)) (ALL-ATOMS (CDR L)))
                  (M-CS 'SEQ (CAR L) (M-CS 'PAR (CDR L) R))
                  F)
              (M-CS 'PAR (CAR L) (M-CS 'SEQ (CDR L) R)))))

(ADD-SHELL PAIR NIL PAIRP ((FIRST (ONE-OF NUMBERP) ZERO) (SECOND (ONE-OF NUMBERP) ZERO)))

(ADD-AXIOM CS-TAKES-PAIRS (REWRITE)
           (IMPLIES (NOT (PAIRP I)) (EQUAL (CS I R) F)))
```

```
(ADD-AXIOM CS-IS-NOT-MIRACLE (REWRITE)
          (EQUAL (CS I F) F))


(PROVE-LEMMA M-CS-IS-NOT-MIRACLE (REWRITE)
             (EQUAL (M-CS FLAG L F) F)
             ((INDUCT (M-CS FLAG L R))))


(PROVE-LEMMA M-CS-IDENTITY (REWRITE)
             (EQUAL (M-CS FLAG (LIST (LIST L)) R) (M-CS FLAG L R))
             ((INDUCT (M-CS FLAG L R))))


(PROVE-LEMMA M-CS-APPEND (REWRITE)
             (IMPLIES (OR (EQUAL FLAG 'SEQ)
                          (AND (EQUAL FLAG 'PAR)
                               (ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))))
               (EQUAL (M-CS FLAG (APPEND L1 L2) R) (M-CS FLAG L1 (M-CS FLAG L2 R))))
             ((INDUCT (M-CS FLAG L1 R))))
```

Trace Transformation Rules

```
(PROVE-LEMMA M-CS-TOTALLY-IND (REWRITE)
             (IMPLIES (TOTALLY-IND-CS (ALL-ATOMS L)) (EQUAL (M-CS 'PAR L R) (M-CS 'SEQ L R)))
             ((INDUCT (M-CS FLAG L R))))


(PROVE-LEMMA G3i (REWRITE)
             (IMPLIES (ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
                      (EQUAL (M-CS 'PAR (CONS L1 L2) R) (M-CS 'SEQ (APPEND L1 (LIST L2)) R))))


(PROVE-LEMMA G3ii (REWRITE)
             (IMPLIES (ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
                      (EQUAL (M-CS 'PAR (CONS (APPEND L1 L) L2) R)
                             (M-CS 'SEQ (APPEND L1 (LIST (CONS L L2))) R)))
             ((INDUCT (APPEND L1 L))))


(ADD-AXIOM IND-CS-IMPLIES-COMMUTATIVITY (REWRITE)
          (IMPLIES (IND-CS I J) (EQUAL (CS J (CS I R)) (CS I (CS J R)))))


(PROVE-LEMMA IS-IND-CS-IMPLIES-COMMUTATIVITY (REWRITE)
             (IMPLIES (IS-IND-CS I (ALL-ATOMS L))
                      (EQUAL (CS I (M-CS FLAG L R)) (M-CS FLAG L (CS I R))))
             ((INDUCT (M-CS FLAG L R))))


(PROVE-LEMMA ARE-IND-CS-IMPLIES-COMMUTATIVITY (REWRITE)
             (IMPLIES (ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
                      (EQUAL (M-CS FLAG1 L1 (M-CS FLAG2 L2 R)) (M-CS FLAG2 L2 (M-CS FLAG1 L1 R))))
             ((INDUCT (M-CS FLAG1 L1 R))))
```

```
(PROVE-LEMMA ARE-IND-CS-IMPLIES-COMMUTATIVITY-SEQ (REWRITE)
             (IMPLIES (ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
                      (EQUAL (M-CS 'SEQ L1 (M-CS 'SEQ L2 R)) (M-CS 'SEQ L2 (M-CS 'SEQ L1 R))))
             ((USE (ARE-IND-CS-IMPLIES-COMMUTATIVITY (FLAG1 'SEQ) (FLAG2 'SEQ)))))
```

Theory of Non-Overlap

```
(DEFN NO-OVERLAP (I J)
      (AND (PAIRP I) (PAIRP J)
           (NOT (EQUAL (FIRST I) (FIRST J)))
           (NOT (EQUAL (FIRST I) (SECOND J)))
           (NOT (EQUAL (SECOND I) (FIRST J)))
           (NOT (EQUAL (SECOND I) (SECOND J)))))

(ADD-AXIOM NO-OVERLAP-IND-CS (REWRITE)
           (IMPLIES (NO-OVERLAP I J) (IND-CS I J)))

(DEFN HAS-NO-OVERLAP (I L)
      (IF (NLISTP L)
          (IF (EQUAL L NIL)
              T
              (NO-OVERLAP I L))
          (AND (NO-OVERLAP I (CAR L)) (HAS-NO-OVERLAP I (CDR L)))))

(PROVE-LEMMA HAS-NO-OVERLAP-IS-IND-CS (REWRITE)
             (IMPLIES (HAS-NO-OVERLAP I L) (IS-IND-CS I L)))

(DEFN HAVE-NO-OVERLAP (L1 L2)
      (IF (NLISTP L1)
          (IF (EQUAL L1 NIL)
              T
              (HAS-NO-OVERLAP L1 L2))
          (AND (HAS-NO-OVERLAP (CAR L1) L2) (HAVE-NO-OVERLAP (CDR L1) L2))))

(PROVE-LEMMA HAVE-NO-OVERLAP-ARE-IND-CS (REWRITE)
             (IMPLIES (HAVE-NO-OVERLAP L1 L2) (ARE-IND-CS L1 L2)))

(DEFN TOTALLY-NO-OVERLAP (L)
      (IF (NLISTP L)
          T
          (AND (HAS-NO-OVERLAP (CAR L) (CDR L)) (TOTALLY-NO-OVERLAP (CDR L)))))

(PROVE-LEMMA TOTALLY-NO-OVERLAP-TOTALLY-IND-CS (REWRITE)
             (IMPLIES (TOTALLY-NO-OVERLAP L) (TOTALLY-IND-CS L)))
```

```
(PROVE-LEMMA HAS-NO-OVERLAP-APPEND (REWRITE)
             (IMPLIES (AND (HAS-NO-OVERLAP I L1) (HAS-NO-OVERLAP I L2))
                      (HAS-NO-OVERLAP I (APPEND L1 L2))))

(PROVE-LEMMA HAVE-NO-OVERLAP-NIL (REWRITE)
             (HAVE-NO-OVERLAP L NIL))

(PROVE-LEMMA HAVE-NO-OVERLAP-APPEND-RIGHT (REWRITE)
             (IMPLIES (AND (HAVE-NO-OVERLAP L1 L2) (HAVE-NO-OVERLAP L1 L3))
                      (HAVE-NO-OVERLAP L1 (APPEND L2 L3))))

(PROVE-LEMMA HAVE-NO-OVERLAP-APPEND-LEFT (REWRITE)
             (IMPLIES (AND (HAVE-NO-OVERLAP L1 L3) (HAVE-NO-OVERLAP L2 L3))
                      (HAVE-NO-OVERLAP (APPEND L1 L2) L3)))

(PROVE-LEMMA TOTALLY-NO-OVERLAP-APPEND (REWRITE)
             (IMPLIES (AND (TOTALLY-NO-OVERLAP L1) (TOTALLY-NO-OVERLAP L2) (HAVE-NO-OVERLAP L1 L2))
                      (TOTALLY-NO-OVERLAP (APPEND L1 L2))))
```

APPLICATION: TRACE TRANSFORMATION OF THE BITONIC SORT

Algebraic Prerequisites

```
(PROVE-LEMMA ZEROP-PLUS-1 (REWRITE)
             (IMPLIES (ZEROP Y) (EQUAL (PLUS X Y) (FIX X))))

(PROVE-LEMMA ZEROP-PLUS-2 (REWRITE)
             (IMPLIES (ZEROP X) (EQUAL (PLUS X Y) (FIX Y))))

(PROVE-LEMMA SELF-DIFFERENCE (REWRITE)
             (EQUAL (DIFFERENCE X X) 0))

(PROVE-LEMMA PLUS-ASSOCIATIVITY (REWRITE)
             (EQUAL (PLUS X (PLUS Y Z)) (PLUS (PLUS X Y) Z)))

(PROVE-LEMMA PLUS-ELIMINATION (REWRITE)
             (EQUAL (DIFFERENCE (PLUS X Y) Y) (FIX X)))

(PROVE-LEMMA PLUS-DIFFERENCE-EXCHANGE-1 (REWRITE)
             (IMPLIES (LEQ Z X) (EQUAL (PLUS (DIFFERENCE X Z) Y) (DIFFERENCE (PLUS X Y) Z))))

(PROVE-LEMMA PLUS-DIFFERENCE-EXCHANGE-2 (REWRITE)
             (IMPLIES (LEQ Z Y) (EQUAL (PLUS X (DIFFERENCE Y Z)) (DIFFERENCE (PLUS X Y) Z))))
```

```
(PROVE-LEMMA LEQ-DIFFERENCE-IS-ZERO (REWRITE)
             (IMPLIES (LEQ X Y) (EQUAL (DIFFERENCE X Y) 0)))

(PROVE-LEMMA DIFFERENCE-REWRITE-1 (REWRITE)
             (IMPLIES (LEQ Z Y)
                      (EQUAL (DIFFERENCE X (DIFFERENCE Y Z))
                             (IF (LESSP X Y)
                                 (DIFFERENCE (PLUS X Z) Y)
                                 (PLUS (DIFFERENCE X Y) Z)))))

(PROVE-LEMMA DIFFERENCE-REWRITE-2 (REWRITE)
             (IMPLIES (LEQ Y X)
                      (EQUAL (DIFFERENCE (DIFFERENCE X Y) Z) (DIFFERENCE (DIFFERENCE X Z) Y))))

(PROVE-LEMMA PLUS-DIFFERENCE-ELIMINATION (REWRITE)
             (IMPLIES (LESSP Y Z)
                      (EQUAL (DIFFERENCE (DIFFERENCE (PLUS X Z) Y) Z) (DIFFERENCE X Y))))

(PROVE-LEMMA ZERO-DIVIDED-BY-ALL (REWRITE)
             (EQUAL (REMAINDER 0 X) 0))

(PROVE-LEMMA SELF-DIVISION (REWRITE)
             (EQUAL (REMAINDER X X) 0))

(PROVE-LEMMA NOT-DIVIDED-IS-NOT-EQUAL (REWRITE)
             (IMPLIES (NOT (EQUAL (REMAINDER X Y) 0)) (NOT (EQUAL X Y))))

(PROVE-LEMMA SUBSTRACT-DIVISORS (REWRITE)
             (IMPLIES (LEQ Y X)
                      (EQUAL (REMAINDER (PLUS X X) Y)
                             (REMAINDER (DIFFERENCE (DIFFERENCE (PLUS X X) Y) Y) Y))))

(PROVE-LEMMA DIVIDED-DOUBLE (REWRITE)
             (IMPLIES (EQUAL (REMAINDER X Y) 0) (EQUAL (REMAINDER (PLUS X X) Y) 0)))

(PROVE-LEMMA DIVISOR-ELIMINATION (REWRITE)
             (EQUAL (REMAINDER (PLUS X Y) Y) (REMAINDER X Y)))

(PROVE-LEMMA ADD-DIVIDEND (REWRITE)
             (IMPLIES (AND (NOT (EQUAL (REMAINDER X Z) 0)) (EQUAL (REMAINDER Y Z) 0))
                      (NOT (EQUAL (REMAINDER (PLUS X Y) Z) 0))))

(PROVE-LEMMA DIVIDEND-ELIMINATION (REWRITE)
             (IMPLIES (AND (EQUAL (REMAINDER Y Z) 0) (EQUAL (REMAINDER (PLUS X Y) Z) 0))
                      (EQUAL (REMAINDER X Z) 0)))
```

```
(PROVE-LEMMA DIVIDED-ELIMINATION (REWRITE)
             (IMPLIES (AND (EQUAL (REMAINDER (DIFFERENCE X Y) Z) 0)
                           (EQUAL (REMAINDER (DIFFERENCE Y X) Z) 0)
                           (EQUAL (REMAINDER Y Z) 0))
                      (EQUAL (REMAINDER X Z) 0))
             ((DISABLE PLUS-DIFFERENCE-EXCHANGE-1)))

(PROVE-LEMMA DIVIDED-ELIMINATION-1 (REWRITE)
             (IMPLIES (AND (EQUAL (REMAINDER (DIFFERENCE X (PLUS Y W)) Z) 0)
                           (EQUAL (REMAINDER (DIFFERENCE (PLUS Y W) X) Z) 0)
                           (EQUAL (REMAINDER W Z ) 0))
                      (EQUAL (REMAINDER (DIFFERENCE X Y) Z) 0)))

(PROVE-LEMMA DIVIDED-ELIMINATION-2 (REWRITE)
             (IMPLIES (AND (EQUAL (REMAINDER (DIFFERENCE X (PLUS Y W)) Z) 0)
                           (EQUAL (REMAINDER (DIFFERENCE (PLUS Y W) X) Z) 0)
                           (EQUAL (REMAINDER W Z ) 0))
               (EQUAL (REMAINDER (DIFFERENCE Y X) Z) 0)))

(PROVE-LEMMA NOT-DIVIDED-1 (REWRITE)
             (IMPLIES (AND (NUMBERP X) (NUMBERP Y) (NOT (EQUAL X Y)) (LESSP X Z) (LESSP Y Z))
                      (OR (NOT (EQUAL (REMAINDER (DIFFERENCE X Y) Z) 0))
                          (NOT (EQUAL (REMAINDER (DIFFERENCE Y X) Z) 0)))))

(PROVE-LEMMA NOT-DIVIDED-2 (REWRITE)
             (IMPLIES (AND (NUMBERP X) (NUMBERP Y) (NOT (EQUAL X Y)) (LESSP X Z) (LESSP Y Z))
                      (OR (NOT (EQUAL (REMAINDER (DIFFERENCE X (PLUS Y Z)) Z) 0))
                          (NOT (EQUAL (REMAINDER (DIFFERENCE (PLUS Y Z) X) Z) 0)))))

(PROVE-LEMMA LESSP-QUOTIENT (REWRITE)
             (IMPLIES (LESSP 0 N) (LESSP (QUOTIENT N 2) N)))
```

Trace Definitions of the Bitonic Sort

```
(DEFN SEGMENT (BTM STEP LENG)
      (IF (ZEROP LENG)
         NIL
         (CONS (PAIR BTM (PLUS BTM STEP))
               (SEGMENT (PLUS BTM (PLUS STEP STEP)) STEP (SUB1 LENG)))))

(DEFN SORT (BASE STEP LENG)
      (IF (OR (ZEROP LENG) (EQUAL LENG 1))
          NIL
          (APPEND (SORT BASE (PLUS STEP STEP) (QUOTIENT LENG 2))
                  (APPEND (SORT (PLUS BASE STEP) (PLUS STEP STEP) (QUOTIENT LENG 2))
                          (SEGMENT BASE STEP (QUOTIENT LENG 2))))))
```

```
(DEFN TAU (N)
      (SORT 0 1 (ADD1 N)))


(DEFN GEN-SEQ-HEAD (SEQ-HEAD STEP)
      (IF (NLISTP SEQ-HEAD)
          NIL
          (CONS (CAR SEQ-HEAD)
                (CONS (PLUS (CAR SEQ-HEAD) STEP)
                      (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP)))))


(DEFN SAME-LEVEL (SEQ-HEAD STEP LENG)
      (IF (NLISTP SEQ-HEAD)
          NIL
          (APPEND (SEGMENT (CAR SEQ-HEAD) STEP (QUOTIENT LENG 2))
                  (SAME-LEVEL (CDR SEQ-HEAD) STEP LENG))))


(DEFN PAR-CMDS (SEQ-HEAD STEP LENG)
      (IF (OR (ZEROP LENG) (EQUAL LENG 1))
          NIL
          (APPEND (PAR-CMDS (GEN-SEQ-HEAD SEQ-HEAD STEP) (PLUS STEP STEP) (QUOTIENT LENG 2))
                  (LIST (SAME-LEVEL SEQ-HEAD STEP LENG)))))


(DEFN TAU~ (N)
      (PAR-CMDS (LIST 0) 1 (ADD1 N)))


(DEFN APPEND-LEVELS (SEQ-HEAD STEP LENG)
      (IF (OR (ZEROP LENG) (EQUAL LENG 1))
          NIL
          (APPEND (APPEND-LEVELS (GEN-SEQ-HEAD SEQ-HEAD STEP) (PLUS STEP STEP) (QUOTIENT LENG 2))
                  (SAME-LEVEL SEQ-HEAD STEP LENG))))


(DEFN TAU' (N)
      (APPEND-LEVELS (LIST 0) 1 (ADD1 N)))


(DEFN SUBTREES (SEQ-HEAD STEP LENG)
      (IF (NLISTP SEQ-HEAD)
          NIL
          (APPEND (SORT (CAR SEQ-HEAD) STEP LENG) (SUBTREES (CDR SEQ-HEAD) STEP LENG))))
```

Trace Transformation Prerequisites

```
(DEFN SEQ-HEADP (SEQ-HEAD STEP)
      (IF (NLISTP SEQ-HEAD)
          T
          (AND (NOT (ZEROP STEP))
               (NUMBERP (CAR SEQ-HEAD))
               (LESSP (CAR SEQ-HEAD) STEP)
               (NOT (MEMBER (CAR SEQ-HEAD) (CDR SEQ-HEAD)))
               (SEQ-HEADP (CDR SEQ-HEAD) STEP))))


(PROVE-LEMMA GEN-SEQ-HEAD-IS-NOT-MEMBER (REWRITE)
             (IMPLIES (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP)
                      (AND (NOT (MEMBER CAR-SEQ-HEAD (GEN-SEQ-HEAD CDR-SEQ-HEAD STEP)))
                           (NOT (MEMBER (PLUS CAR-SEQ-HEAD STEP) (GEN-SEQ-HEAD CDR-SEQ-HEAD STEP))))))


(PROVE-LEMMA GEN-SEQ-HEAD-IS-SEQ-HEADP (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                      (SEQ-HEADP (GEN-SEQ-HEAD SEQ-HEAD STEP) (PLUS STEP STEP)))
             ((DISABLE SUBSTRACT-DIVISORS)))


(PROVE-LEMMA ALL-ATOMS-SEGMENT (REWRITE)
             (EQUAL (ALL-ATOMS (SEGMENT BTM STEP LENG)) (SEGMENT BTM STEP LENG)))


(PROVE-LEMMA ALL-ATOMS-SORT (REWRITE)
             (EQUAL (ALL-ATOMS (SORT BASE STEP LENG)) (SORT BASE STEP LENG)))


(PROVE-LEMMA ALL-ATOMS-SAME-LEVEL (REWRITE)
             (EQUAL (ALL-ATOMS (SAME-LEVEL SEQ-HEAD STEP LENG)) (SAME-LEVEL SEQ-HEAD STEP LENG)))


(PROVE-LEMMA ALL-ATOMS-SUBTREES (REWRITE)
             (EQUAL (ALL-ATOMS (SUBTREES SEQ-HEAD STEP LENG)) (SUBTREES SEQ-HEAD STEP LENG)))


(PROVE-LEMMA NO-OVERLAP-PAIRS-1 (REWRITE)
             (IMPLIES (AND (OR (NOT (EQUAL (REMAINDER (DIFFERENCE BTM1 BTM2) STEP2) 0))
                               (NOT (EQUAL (REMAINDER (DIFFERENCE BTM2 BTM1) STEP2) 0)))
                           (EQUAL (REMAINDER STEP1 STEP2) 0))
                      (NO-OVERLAP (PAIR BTM1 (PLUS BTM1 STEP1)) (PAIR BTM2 (PLUS BTM2 STEP2))))
             ((DISABLE PLUS REMAINDER)))


(PROVE-LEMMA HAS-NO-OVERLAP-SEGMENT-1 (REWRITE)
             (IMPLIES (AND (OR (NOT (EQUAL (REMAINDER (DIFFERENCE BTM1 BTM2) STEP2) 0))
                               (NOT (EQUAL (REMAINDER (DIFFERENCE BTM2 BTM1) STEP2) 0)))
                           (EQUAL (REMAINDER STEP1 STEP2) 0))
                      (HAS-NO-OVERLAP (PAIR BTM1 (PLUS BTM1 STEP1)) (SEGMENT BTM2 STEP2 LENG2)))
             ((DISABLE NO-OVERLAP PLUS-ASSOCIATIVITY)))
```

```
(PROVE-LEMMA HAVE-NO-OVERLAP-SEGMENT-1 (REWRITE)
             (IMPLIES (AND (OR (NOT (EQUAL (REMAINDER (DIFFERENCE BTM1 BTM2) STEP2) 0))
                                (NOT (EQUAL (REMAINDER (DIFFERENCE BTM2 BTM1) STEP2) 0)))
                            (EQUAL (REMAINDER STEP1 STEP2) 0))
                       (HAVE-NO-OVERLAP (SEGMENT BTM1 STEP1 LENG1) (SEGMENT BTM2 STEP2 LENG2)))
             ((DISABLE HAS-NO-OVERLAP PLUS-ASSOCIATIVITY)))

(PROVE-LEMMA HAS-NO-OVERLAP-SEGMENT-2 (REWRITE)
             (IMPLIES (AND (LESSP X BTM) (LESSP Y BTM))
                       (HAS-NO-OVERLAP (PAIR X Y) (SEGMENT BTM STEP LENG))))

(PROVE-LEMMA TOTALLY-IND-SEGMENT-1 (REWRITE)
             (IMPLIES (NOT (ZEROP STEP)) (TOTALLY-NO-OVERLAP (SEGMENT BTM STEP LENG))))

(PROVE-LEMMA SEQ-HEADP-SEGMENT-HAVE-NO-OVERLAP-1 (REWRITE)
             (IMPLIES (AND (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP) (LISTP CDR-SEQ-HEAD))
                       (HAVE-NO-OVERLAP (SEGMENT CAR-SEQ-HEAD STEP LENG1)
                                        (SEGMENT (CAR CDR-SEQ-HEAD) STEP LENG2)))
             ((USE (NOT-DIVIDED-1 (X CAR-SEQ-HEAD) (Y (CAR CDR-SEQ-HEAD)) (Z STEP)))))

(PROVE-LEMMA SEGMENT-AND-SAME-LEVEL-HAVE-NO-OVERLAP-1 (REWRITE)
             (IMPLIES (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP)
                       (HAVE-NO-OVERLAP (SEGMENT CAR-SEQ-HEAD STEP LENG1)
                                        (SAME-LEVEL CDR-SEQ-HEAD STEP LENG2)))
             ((INDUCT (SAME-LEVEL CDR-SEQ-HEAD STEP LENG2))))

(PROVE-LEMMA SEGMENT-AND-SAME-LEVEL-HAVE-NO-OVERLAP (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                       (HAVE-NO-OVERLAP (SEGMENT (CAR SEQ-HEAD) STEP LENG1)
                                        (SAME-LEVEL (CDR SEQ-HEAD) STEP LENG2))))

(PROVE-LEMMA SAME-LEVEL-IS-TOTALLY-NO-OVERLAP (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                       (TOTALLY-NO-OVERLAP (SAME-LEVEL SEQ-HEAD STEP LENG))))

(PROVE-LEMMA SORT-AND-SEGMENT-HAVE-NO-OVERLAP (REWRITE)
             (IMPLIES (AND (OR (NOT (EQUAL (REMAINDER (DIFFERENCE BTM1 BTM2) STEP2) 0))
                                (NOT (EQUAL (REMAINDER (DIFFERENCE BTM2 BTM1) STEP2) 0)))
                            (EQUAL (REMAINDER STEP1 STEP2) 0))
                       (HAVE-NO-OVERLAP (SORT BTM1 STEP1 LENG1) (SEGMENT BTM2 STEP2 LENG2)))
             ((DISABLE SEGMENT HAS-NO-OVERLAP)))

(PROVE-LEMMA SORT-AND-SEGMENT-HAVE-NO-OVERLAP-1 (REWRITE)
             (IMPLIES (AND (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP)
                            (LISTP CDR-SEQ-HEAD))
                       (HAVE-NO-OVERLAP (SORT (CAR CDR-SEQ-HEAD) (PLUS STEP STEP) LENG1)
                                        (SEGMENT CAR-SEQ-HEAD STEP LENG2)))
             ((USE (NOT-DIVIDED-1 (X CAR-SEQ-HEAD) (Y (CAR CDR-SEQ-HEAD)) (Z STEP)))))
```

```
(PROVE-LEMMA SORT-AND-SEGMENT-HAVE-NO-OVERLAP-2 (REWRITE)
             (IMPLIES (AND (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP)
                           (LISTP CDR-SEQ-HEAD))
                      (HAVE-NO-OVERLAP (SORT (PLUS (CAR CDR-SEQ-HEAD) STEP) (PLUS STEP STEP) LENG1)
                                       (SEGMENT CAR-SEQ-HEAD STEP LENG2)))
             ((USE (NOT-DIVIDED-2 (X CAR-SEQ-HEAD) (Y (CAR CDR-SEQ-HEAD)) (Z STEP)))))

(PROVE-LEMMA SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP-1 (REWRITE)
             (IMPLIES (SEQ-HEADP (CONS CAR-SEQ-HEAD CDR-SEQ-HEAD) STEP)
                      (HAVE-NO-OVERLAP (SUBTREES (GEN-SEQ-HEAD CDR-SEQ-HEAD STEP)
                                                 (PLUS STEP STEP) LENG1)
                                       (SEGMENT CAR-SEQ-HEAD STEP LENG2)))
             ((DISABLE SEGMENT)))

(PROVE-LEMMA SUBTREES-AND-SEGMENT-HAVE-NO-OVERLAP (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                      (HAVE-NO-OVERLAP (SUBTREES (GEN-SEQ-HEAD (CDR SEQ-HEAD) STEP)
                                                 (PLUS STEP STEP) LENG1)
                                       (SEGMENT (CAR SEQ-HEAD) STEP LENG2))))
```

Trace Transformation

```
(PROVE-LEMMA SUBTREES-COMMUTATIVITY (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                      (EQUAL (M-CS 'SEQ
                                   (SUBTREES (GEN-SEQ-HEAD SEQ-HEAD STEP)
                                             (PLUS STEP STEP) (QUOTIENT LENG 2))
                                   (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))
                             (M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R)))
             ((DISABLE SEGMENT-AND-SAME-LEVEL-HAVE-NO-OVERLAP
                       SEGMENT-AND-SAME-LEVEL-HAVE-NO-OVERLAP-1
                       NOT-DIVIDED-IS-NOT-EQUAL)))

(PROVE-LEMMA SUBTREES-NIL (REWRITE)
             (IMPLIES (OR (ZEROP LENG) (EQUAL LENG 1)) (EQUAL (SUBTREES SEQ-HEAD STEP LENG) NIL)))

(DEFN INDUCTION-SCHEME (SEQ-HEAD STEP LENG R)
      (IF (OR (ZEROP LENG) (EQUAL LENG 1))
          NIL
          (INDUCTION-SCHEME (GEN-SEQ-HEAD SEQ-HEAD STEP)
                            (PLUS STEP STEP) (QUOTIENT LENG 2)
                            (M-CS 'SEQ (SAME-LEVEL SEQ-HEAD STEP LENG) R))))
```

```
(PROVE-LEMMA APPEND-LEVELS-EQ-SUBTREES (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                      (EQUAL (M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG) R)
                             (M-CS 'SEQ (SUBTREES SEQ-HEAD STEP LENG) R)))
             ((INDUCT (INDUCTION-SCHEME SEQ-HEAD STEP LENG R))
              (DISABLE NOT-DIVIDED-IS-NOT-EQUAL)))

(PROVE-LEMMA TAU'-EQ-TAU (REWRITE)
             (EQUAL (M-CS 'SEQ (TAU' N) R) (M-CS 'SEQ (TAU  N) R)))

(PROVE-LEMMA PAR-CMDS-MERGE (REWRITE)
             (IMPLIES (SEQ-HEADP SEQ-HEAD STEP)
                      (EQUAL (M-CS 'SEQ (PAR-CMDS SEQ-HEAD STEP LENG) R)
                             (M-CS 'SEQ (APPEND-LEVELS SEQ-HEAD STEP LENG) R)))
             ((INDUCT (INDUCTION-SCHEME SEQ-HEAD STEP LENG R))
              (DISABLE APPEND-LEVELS-EQ-SUBTREES NOT-DIVIDED-IS-NOT-EQUAL)))

(PROVE-LEMMA TAU~-EQ-TAU' (REWRITE)
             (EQUAL (M-CS 'SEQ (TAU~ N) R) (M-CS 'SEQ (TAU' N) R)))

(PROVE-LEMMA TAU~-EQ-TAU (REWRITE)
             (EQUAL (M-CS 'SEQ (TAU~ N) R) (M-CS 'SEQ (TAU  N) R))
             ((DISABLE TAU TAU~ TAU')))
```

TR-84-30 ERRATA

p. 8, line 4 of the definition of ARE-IND-CS: `(IF L1=NIL`

p. 9, line 2: `(TOTALLY-IND-CS L)`

p. 16, the following expansion makes our illustrations clearer:

> ... let us assert:
>
> `(SEQ-HEADP '(0 2 1 3) 4) = T`
>
> From this fact, theorem GEN-SEQ-HEAD-IS-SEQ-HEADP, and the definition of SEQ-HEADP we can conclude the premises of the induction hypotheses of all transformation theorems that we are going to illustrate with this example. The proof of SUBTREES-COMMUTATIVITY is based on an induction suggested by the recursive definition of function GEN-SEQ-HEAD. With validity of the premise, the induction hypothesis reduces to: ...

p. 23, comment of the third equation of the proof of PAR-CMDS-MERGE:
apply SAME-LEVEL-IS-TOTALLY-NO-OVERLAP, not PAR-CMDS-IS-TOTALLY-NO-OVERLAP.

p. 24, Point 5: `(PROVE-LEMMA name types term hints)`

p. 24, explanation of (DISABLE ev_1 ... ev_n): Replace a_n by ev_n.

The whole report: references to [7] apply equally to [6].