AUTOMATED DEDUCTION IN PROGRAMMING LANGUAGE SEMANTICS: THE MECHANICAL CERTIFICATION OF PROGRAM TRANSFORMATIONS TO DERIVE CONCURRENCY

Christian Lengauer & Chua-Huang Huang

TR-85-04 January 1985

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

A preliminary version of this report was presented at the NSF-SERC Seminar on Concurrency at Carnegie-Mellon University, Pittsburgh, Pennsylvania, 9-11 July 1984.

Abstract

This is an attempt to combine the two research areas of programming methodology and automated theorem proving. The potential for automation of a programming methodology is investigated that supports the compile-time derivation of concurrency in imperative programs. In this methodology, concurrency is identified by the declaration of certain semantic properties, so-called "semantic relations", of appropriate program parts. Semantic relations can be exploited to transform the sequential execution of the program into a parallel execution. The methodology is applied to a limited domain of programs: sorting networks. A mechanized theory of transformations of sorting networks is presented and observations about the mechanical certification of three transformations are made.

Most mechanical verification systems do not actually use the formal semantics of the programming language but rather employ some ad hoc device which stands between the program to be verified and the formulas to be proved, for instance, an informally derived verification condition generator. The present approach makes the formal semantics available to the deduction engine directly. This increases the burden on the deduction system, but has several distinct advantages. One advantage is higher confidence in the correctness of the verification system itself. Another more important advantage is that the structure of proofs depends solely on the formal semantics of the programming language. This permits reasoning about the semantics itself rather than just reasoning about individual programs.

1. Introduction

This paper is about the feasibility of a research area: *programming methodology*, or the formal derivation of programs. Programming methodology employs techniques of *formal semantics*, i.e., the formal description of the meaning of programming language constructs and *verification*, i.e., the formal proof of programs. Like the formal proof of programs, the formal derivation of programs will be feasible in a software production environment only if it is mechanically supported. Program logics in their present form are technically too intricate to be effectively and reliably applied by hand on a large scale, and it is doubtful that they will become simpler in the future. (This is not to say that the formal derivation and proof of programs by hand is not of considerable academic interest.) The research area that deals with the automation of formal logics is *automated theorem proving*. We would like to contribute to the currently emerging and very important link between programming methodology and automated theorem proving.

The methodology in whose automation we are interested focusses on the static derivation of concurrency in imperative programs [11, 12]. The derivation of concurrency proceeds by a successive compression of the program's executions based on the declaration of certain useful program properties. Most interesting programs contain recursions or loops. The most effective and practical transformations of such programs will also be recursive, and their proofs of correctness will require induction. We are therefore interested in the mechanical treatment of recursion and induction.

The following section contains a brief review of our methodology. Details can be found in [11, 12]. Sect. 3 introduces the class of programs that we explore: sorting networks. We then describe a mechanized theory of trace transformations, its application to several sorting networks, and the challenges we encountered in our mechanical proofs (Sect. 4). We conclude the paper with a discussion of the role that we see for the mechanization of semantic theories (Sect. 5). An appendix contains the executable code of our mechanized theory.

2. The Methodology

Our methodology supports the *derivation* of concurrency *after* the program development. It does not permit the *specification* of concurrency *before* the program development. We view concurrency is an optimization, not as a structural property of the program. The sole purpose of the concurrency that we consider is to accelerate the acquisition of results.

The most common approach to programming in which the derivation of concurrency is divorced from the derivation of the program is data flow programming [1]. A data flow program makes no explicit reference to the order of execution. It is executed on a special machine architecture that follows the sequencing imposed by the data dependencies of the program's variables. Data flow languages are "referentially

transparent": they do not permit the re-assignment of variables. This simplifies the identification of data independencies so much that, commonly, no programmer assistance is needed to identify concurrency. We take the "referentially opaque" approach, i.e., permit the re-assignment of variables and, consequently, require a more complicated data flow analysis. We have to explicitly declare and subsequently exploit data independencies as what we call "semantic relations". In our methodology, the development of programs is divided into two stages:

Stage 1: The development and formal semantic description of a *program* that achieves the desired result. This requires a formal refinement and the declaration of semantic relations. Programs are composed by the usual program combinators, e.g., composition: `S1;S2` (read: "`S2` is applied to the results of `S1`"). Programs do not explicitly address the question of execution order.

Stage 2: The derivation of a fast *execution* of the program produced at Stage 1. (An execution of a program is also called a *trace*.) This is conceptually simple but computationally complex. It involves the computation of execution times and the invocation of semantic relations to transform traces and improve execution time. There are two trace combinators: `S1→S2` (read: "execute `S1` and then `S2`"), and `<S1 S2>` (read: "execute `S1` and `S2` in parallel").

The refinement suggests an easy sequential implementation: replace every composition `S1;S2` by sequential execution `S1→S2`.

Semantic relations are, for instance, the commutativity of the components `S1` and `S2` (written `S1&S2`), and the independence of `S1` and `S2` (written `S1||S2`). `S1` and `S2` are commutative, i.e., `S1&S2` may be declared if the execution of `S1` and then `S2` has the same effect as the execution of `S2` and then `S1`. If `S1&S2` is declared, `S1;S2` may also be implemented by `S2→S1`. `S1` and `S2` are independent, i.e., `S1||S2` may be declared if the execution of `S1` and `S2` in parallel has the same effect as their execution in order. If `S1||S2` is declared, `S1;S2` may also be implemented by `<S1 S2>`. A third semantic relation is the idempotence of component `S` (written `!S`). `S` is idempotent, i.e., `!S` may be declared if `S` has the same effect as `S;S`. If `!S` is declared, we may add to or delete from a sequence of consecutive calls of `S`.

Idempotence helps eliminate superfluous parts of an execution, or duplicate parts of an execution for commutation to appropriate places. Commutativity helps distribute program components to places in the execution where they can be executed in concurrence with others. Independence helps add concurrency. Independence implies commutativity.

We call Stage 1 the *refinement calculus* and Stage 2 the *trace calculus*. Either of the two stages has the potential for automation. We are interested in the mechanical support of Stage 2. Our current focus is the correctness proof of trace transformations. We view trace transformations as theorems of trace equivalences.[1] Induction permits us to certify unbounded transformations by a finite argument.

[1] For an elaboration, see Sect. 3 of [13].

Presently, we accept the source and target trace of a transformation as given and, therefore, leave execution time out of our mechanization. For the certification of trace transformations, we employ Boyer & Moore's induction prover [4] that is based on a mechanized functional logic particularly suitable for program verification [3]. The prover is designed to prove theorems about recursive functions but is not an expert on sorting networks and their trace transformations. By implementing our theory of trace transformations in Boyer & Moore's logic, we attempt to turn their prover into such an expert.

3. The Expository Domain: Sorting Networks

Semantic relations, as defined in our methodology, can be declared for programs in any imperative programming language that has a weakest precondition semantics. For the purpose of our investigation we choose a very simple language. We do not want to complicate our mechanical proofs of trace transformations by unduly complicated semantics of programs and traces. We define the language of *sorting networks* [10]. The general problem that we pursue is to sort an array $a_{0..n}$ of numbers into ascending order in no more time than O(n). The linear time requirement forces us to consider a concurrent execution. In the language of sorting networks, refinements can have the following structure:

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

(2) The *comparator module* `cs(i,j)` accesses an array `a` of numbers. It compares elements a_i and a_j and, if necessary, interchanges them into order. A simpler version of comparator module deals with adjacent elements a_{i-1} and a_i. Instead of writing `cs(i-1,i)`, we shall give simple comparator modules only one argument `cs(i)`. We call sorting networks that are composed of simple comparator modules *simple sorting networks*. The comparator module is of imperative nature, i.e., its implementation requires assignment.

(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.

Sorting networks are well-suited for our methodology because they terminate and only their results, not their behaviors matter. They also have a wide range of applications and are extensively researched. It is important to realize that we are not trying to do research in sorting networks. We chose them as a well-understood first domain in which to test our ideas of automation.

Since we are concerned with the trace calculus of the methodology, we do not dwell on the refinement of programs but accept the particular sorting networks whose trace transformations we want to study as given. We will study three sorting networks: the insertion sort, the odd-even transposition sort, and the bitonic sort [10]. The insertion sort and the odd-even transposition sort can be expressed as simple sorting networks. The bitonic sort expects array `a` already presorted in bitonic order. Let us describe each of the three sorting networks in turn.

3.1. Insertion Sort

The following refinement describes a sorting network that performs an insertion sort:

```
insertion-sort(n):   sort(n)

             sort(0):   skip
{i>0}        sort(i):   sort(i-1); S(i)

             S(0):      skip
{i>0}        S(i):      cs(i); S(i-1)
```

Comparator modules may be declared idempotent. Consecutive applications of the same comparator module do not yield any new results. For `|i-j|>1`, i.e., if `i` and `j` are not "neighbors", `cs(i)` and `cs(j)` are disjoint: they do not share any variables. Components that do not share variables may be declared independent.

$$\begin{array}{lcl} & & \texttt{!cs(i)} \\ \texttt{|i-j|>1} & \Rightarrow & \texttt{cs(i)||cs(j)} \end{array}$$

Note that the prerequisite `|i-j|>1` makes `cs(i)||cs(j)` a semantic rather than syntactic condition. (Semantic declarations can also be qualified with respect to a postcondition. For the underlying theory see [12].)

For, say, a 6-element array (n=5), the refinement suggests the following sequential execution, if we expand components `sort(i)` and `S(i)` (i≤5) of `sort(5)`:

```
tau(5) = cs(1)→cs(2)→cs(1)
                    →cs(3)→cs(2)→cs(1)
                          →cs(4)→cs(3)→cs(2)→cs(1)
                                →cs(5)→cs(4)→cs(3)→cs(2)→cs(1)
```

If we count the number of comparator modules `cs`, `tau(5)` has length 15. In general, `tau(n)` has length `n(n+1)/2`, i.e., is quadratic in `n`. To derive a linear execution, we have to exploit the independence declaration for `sort(n)` and compress `tau(n)` into a trace with concurrency. We have already laid out the sequential trace `tau(5)` in a form which suggests how this can be done. We commute comparator modules in `tau(5)` left, and then merge adjacent modules whose indices differ by 2 into a parallel command:

$$\texttt{tau}^{\sim}\texttt{(5)} = \texttt{cs(1)}\rightarrow\texttt{cs(2)}\rightarrow\left\langle \begin{array}{l}\texttt{cs(1)}\\ \\ \texttt{cs(3)}\end{array}\right\rangle\rightarrow\left\langle \begin{array}{l}\texttt{cs(2)}\\ \\ \texttt{cs(4)}\end{array}\right\rangle\rightarrow\left\langle \begin{array}{l}\texttt{cs(1)}\\ \texttt{cs(3)}\\ \texttt{cs(5)}\end{array}\right\rangle\rightarrow\left\langle \begin{array}{l}\texttt{cs(2)}\\ \\ \texttt{cs(4)}\end{array}\right\rangle\rightarrow\left\langle \begin{array}{l}\texttt{cs(1)}\\ \\ \texttt{cs(3)}\end{array}\right\rangle\rightarrow\texttt{cs(2)}\rightarrow\texttt{cs(1)}$$

If we assume instantaneous initiation and termination of parallel commands (instantaneous forks and joins), this execution is of length 9. In general, `tau~(n)` is of length `2n-1`, i.e., linear in `n`. The degree of concurrency increases as we add inputs. This is a property of all three sorting networks. They are not limited to a fixed number of concurrent actions. However, if only a fixed number `k` of processors is available, the independence declaration may be exploited only to generate a concurrency degree of `k` or less.

Note that the idempotence declaration of comparator modules does not help in the derivation of concurrency for the insertion sort. As we shall see in the next section, array $a_{0..n}$ can be sorted faster than by `tau~(n)`, but not when we start with the refinement of the insertion sort.

3.2. Odd-Even Transposition Sort

The odd-even transposition sort is the simplest possible example of the transformation of a sorting network. Here is the refinement:

```
odd-even-sort(n):     sort(n+1,n)

          sort(0,j):  skip
          sort(1,j):  S(j-1)
{i>1}     sort(i,j):  S(j-1); S(j); sort(i-2,j)

          S(0):       skip
          S(1):       cs(1)
{i>1}     S(i):       cs(i); S(i-2)
```

As a simple sorting network like the insertion sort, the odd-even transposition sort adopts the semantic declarations of the previous section:

$$\begin{array}{lcl} & & \texttt{!cs(i)} \\ \texttt{|i-j|>1} & \Rightarrow & \texttt{cs(i)||cs(j)} \end{array}$$

For, e.g., a 5-element array (`n=4`), the sequential trace suggested by this refinement is:

`tau(4) = cs(3)→cs(1)→cs(4)→cs(2)→cs(3)→cs(1)→cs(4)→cs(2)→cs(3)→cs(1)`

The number of comparator modules in `tau(4)` is 10. In general, `tau(n)` has length `n(n+1)/2`. In every `S(i)`, the indices of all comparator modules differ at least by 2. Thus we can convert each `S(i)` into one parallel command. The resulting parallel trace is:

$$\texttt{tau~(4)} = \left\langle \begin{matrix}\texttt{cs(1)}\\ \texttt{cs(3)}\end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix}\texttt{cs(2)}\\ \texttt{cs(4)}\end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix}\texttt{cs(1)}\\ \texttt{cs(3)}\end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix}\texttt{cs(2)}\\ \texttt{cs(4)}\end{matrix} \right\rangle \rightarrow \left\langle \begin{matrix}\texttt{cs(1)}\\ \texttt{cs(3)}\end{matrix} \right\rangle$$

`tau~(4)` is of length 5. In general, `tau~(n)` is of length `n+1`.

3.3. Bitonic Sort

An array $a_{0..n}$ is in *bitonic order* if $a_0 \geq \ldots \geq a_i \leq \ldots \leq a_n$ for some `i` such that $0 \leq i \leq n$. Let us write array $a_{0..n}$ as a sequence $\langle a_0, a_1, \ldots, a_n \rangle$. The bitonic sorting algorithm sorts such a sequence, if it is already in bitonic order, into non-decreasing 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 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. The bitonic sort is not a simple sorting network. It requires the general comparator module `cs(i,j)`.

The significance of the bitonic sort lies in the fact that we can derive from it a network that sorts arbitrary (not bitonic) sequences.[2] 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. The extension to arbitrary sequences does not add any new issue in our program development. Therefore we shall not address it any further.

The 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⌋);
                                        S(base,step,⌊leng/2⌋)

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

Refinement `sort` performs the bitonic sort as described. Refinement `S` 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.

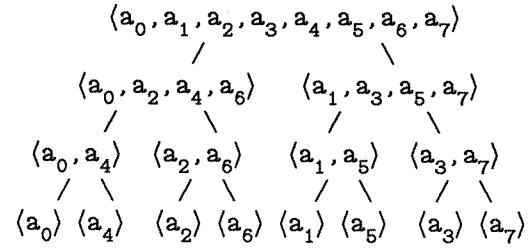
Like simple comparator modules, general comparator modules may be declared idempotent. Also, disjoint comparator modules may be declared independent. General comparator modules $cs(i_1,i_2)$ and $cs(j_1,j_2)$ are disjoint if they do not overlap, i.e., if $i_1 \neq j_1$, $i_1 \neq j_2$, $i_2 \neq j_1$ and $i_2 \neq j_2$.

$$i_1 \neq j_1 \wedge i_1 \neq j_2 \wedge i_2 \neq j_1 \wedge i_2 \neq j_2 \;\Rightarrow\; \begin{matrix} !cs(i_1,i_2) \\ cs(i_1,i_2)\,||\,cs(j_1,j_2) \end{matrix}$$

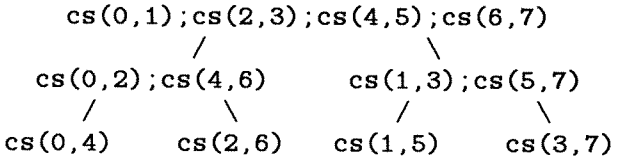
Let us construct a binary tree of bitonic sequences whose root is the entire sequence a, and whose left and right subtrees are recursively constructed by splitting the root into subsequences as prescribed by the bitonic sorting algorithm. We call this tree the *sequence tree* of a. The sequence tree of an 8-element sequence (n=7) is:

---

[2] See Exercise 13 of [10].



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 $\texttt{cs}(i_1,i_2);\texttt{cs}(i_3,i_4);\ldots$, which we shall call a *segment*. The following *segment tree* corresponds to the above sequence tree:



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

Note that, in the refinement of the bitonic sort, segments are represented by calls of S. We can now view the sequential trace tau suggested by the refinement as the post-order traversal of segments in the segment tree:

```
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)
```

tau(7) has length 12. In general, 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 earlier. With $n=2^k-1$, trace tau(n) has a length of order O(n log n).

Observe that any two distinct segments x and y in the segment tree which are not in an ascendant/descendant relationship do not access 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)>
```

tau~(7) is of length 3, with a concurrency degree of 4. In general, tau~(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).

4. The Mechanical Correctness Proof of Trace Transformations

We have implemented our theory of trace transformations in Boyer & Moore's mechanized logic [3]. Boyer & Moore express terms of first-order predicate logic in a LISP-like functional form.[3] 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

(NUMBERP X) ⇒ X < 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 (PAIR i_1 i_2) or, abbreviated, (i_1: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 our proofs of transformations of sorting networks. There are several ways to make use of a previously established lemma in subsequent proofs. Our proofs employ 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 trace transformation theorems for sorting networks in Boyer & Moore's logic. We present the theory of general sorting networks, which is a generalization of the theory of simple sorting networks presented in a previous paper [13].[4] An appendix contains the executable code. We advise readers without experience in Boyer & Moore's logic to consult [3] before studying the appendix.

---

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

[4] In the theory of simple sorting networks, the comparator module is represented by a number, not a pair of numbers.

4.1. Trace Representation

Our goal is to prove the semantic equivalence of traces `tau(n)` and `tau~(n)`. We represent a trace as 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 of the bitonic sort

```
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 $cs(i_1, i_2)$.

4.2. Trace Semantics

We give traces weakest precondition semantics [12]. 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 [7]. 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). But we must implement the weakest precondition of the comparator module.

Our methodology divides the development of programs into two stages. Stage 1, the refinement calculus, is concerned with the *derivation* of program semantics, i.e., the derivation of a refinement. Stage 2, the trace calculus, is concerned with the *preservation* of program semantics, i.e., the transformation of sequential executions into concurrent executions. We are implementing Stage 2, and are therefore only interested in the equality of weakest preconditions, not in their actual values. If the inside of a program

component is never affected by a trace transformation, we need not spell out its semantics but may provide it as a "black box". In Boyer & Moore's logic, a black box is represented by a function that has been *declared* (without a function body) rather than *defined* (with a function body).

The inside of comparator modules will never be subject to transformation. Therefore 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 function CS is declared, not defined, we must provide by axiom some essential information about CS that is not evident from the declaration. 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 weakest 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 [7]) for comparator modules:

Axiom CS-IS-NOT-MIRACLE:

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

Axiom CS-IS-NOT-MIRACLE states that the weakest 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:

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

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 of CS as prescribed by trace L. Besides 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. Since we do not provide the complete semantics of comparator modules, we cannot provide a complete characterization of their independence [12]. We express independence, 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 all comparator modules of a trace L:

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 of a trace L1 with all comparator modules of a trace L2:

Definition

```
(ARE-IND-CS L1 L2)
   =
(IF (NOT (LISTP L1))
    (IF L1=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 mutual independence of all comparator modules of a trace L:

Definition

```
(TOTALLY-IND-CS L)
   =
(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 lists that end with any atom. One theorem that we use often in our proofs is the composition rule of weakest preconditions [7]:

Lemma M-CS-APPEND:

```
(FLAG='SEQ)
   ⇒  (   (M-CS FLAG (APPEND L1 L2) R)
        = (M-CS FLAG L1 (M-CS FLAG L2 R)) )
```

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

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

4.3. Trace Transformation

We are now at the point where we can begin formulating the transformation of **tau** into **tau~** in Boyer & Moore's logic. We need transformation rules that express commutations and parallel merges of independent program components. All theorems except the first, which does not require induction, are proved by structural induction, i.e., by an induction that mirrors the recursive definition of weakest precondition generator M-CS.

We provide several theorems for parallel merges. Two theorems express transformation rules (G3i) and (G3ii) of Sect. 5.2 of [12]:

Lemma G3i:

```
(ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
   ⇒  (   (M-CS 'PAR (CONS L1 L2) R)
        = (M-CS 'SEQ (APPEND L1 (LIST L2)) R) )
```

Lemma G3ii:

```
(ARE-IND-CS (ALL-ATOMS L1) (ALL-ATOMS L2))
   ⇒  (   (M-CS 'PAR (CONS (APPEND L1 L) L2) R)
        = (M-CS 'SEQ (APPEND L1 (LIST (CONS L L2))) R) )
```

Phrased abstractly, G3i states the semantic equivalence of traces <L1 L2> and L1→L2, G3ii of traces <L1→L L2> and L1→<L L2>, provided L1 and L2 are independent. Our third theorem is:

Lemma M-CS-TOTALLY-IND:

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

Both G3ii and M-CS-TOTALLY-IND are generalizations of G3i.

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)) )
```

Just as we cannot compute weakest preconditions of comparator modules with declared function CS, we cannot determine their independence with declared function IND-CS. While, in our applications of the theory, 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. 3 that two comparator modules are independent if they do not access common array elements, i.e., if they do not overlap.[6] 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))
```

[6] We called simple comparator modules with this property "non-neighbors".

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 of NO-OVERLAP substituted for calls of IND-CS. Theorems stating that each of the three non-overlap functions implies its respective independence counterpart can be proved from axiom NO-OVERLAP-ARE-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 basic theory is not fully represented in the mechanized logic: we introduced two declared (not defined) functions CS and IND-CS. To be able to reason about them, we had to formulate five axioms. They reflect the properties of comparator modules and their independence that we are willing to accept without certification. With suitable definitions of CS and IND-CS these axioms could be converted into theorems. It is important that every axiomatic assumption is fully understood. An inconsistency in an axiom is not recognized by the prover and puts the entire mechanized theory in jeopardy!

The next section summarizes our applications of the basic theory: the transformations of the insertion, odd-even transposition, and bitonic sort.

4.4. Applications

Ideally, we would like to submit to the prover nothing else but an application theorem - ours are of the form:

Lemma: TAU-MAIN:

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

where TAU and TAU~ are defined appropriately - and have it certified without any further input. However, no existing prover is expert enough in the theory of trace transformations of sorting networks to accomplish such a proof on its own. To educate the prover, we must implement our theory on it, i.e., express the theory in the mechanized logic, and have it certified and at disposal for further proofs.

Even with the basic theory in place and after proper definition of the initial trace TAU and the final trace TAU~, the work required to make the proof of an application TAU-MAIN succeed is substantial. Essentially, we have to communicate our proof strategy to the prover. Where the transformation consists of several steps, the prover may have to be informed about each individual step. For instance, since we can commute at any place where we can merge (remember that independence implies commutativity), we must tell the prover what transformation we prefer: commutation or merge. Our transformations of the

insertion sort and the bitonic sort each consist of two steps: one of commutations and one of merges. The transformation of the odd-even sort consists of only one step of merges. For every step of the transformation, the trace parts that are manipulated must be identified, and their independence must be established. This generally involves educating the prover about useful facts of number theory. For our simple sorting networks, we had to tell the prover about properties of maximization, for our general sorting network about properties of division. Establishing these prerequisites before the proof of the application theorem is the most tedious aspect of a mechanical certification.[7] For an effective use of a mechanized theory in many applications, clean and widely applicable proof strategies are of central importance.

BASIC THEORY		trace semantics trace transformation rules independence criterion		
		insertion sort	*odd-even sort*	*bitonic sort*
	algebraic prerequisites	maximization	maximization	division
APPLICATION	*transformation strategy*	1st step: commute 2nd step: merge	one step: merge	1st step: commute 2nd step: merge
	auxiliary lemmas	see [13]		see [9]
	main theorem	TAU-MAIN	TAU-MAIN	TAU-MAIN

The previous table displays the overall proof structure of our three applications. The proof of the insertion sort in the theory of simple sorting networks is documented in [13]. We have since converted it to the theory of general sorting networks. [9] describes the proof of the bitonic sort.

While the basic theory may contain some declared functions and axioms (and ours does), the application part of the proof should not (and ours do not). That is, with respect to the basic theory, applications should be completely certified.

4.5. Discussion

By its very name, the area of automated theorem proving invites high expectations: the hope is kindled that, whenever the human prover seems lost or uncertain in a proof, the mechanism will take over and guide him along. A presently more fitting name would be *automated proof checking*: the human has to conceive and carry out the proof; but he can count on a mechanical certification of his proof steps, if these steps are chosen appropriately. In order to make the mechanical certification succeed, the human

[7] We have concentrated our efforts on the implementation of the theory of trace transformations, not the theory of natural arithmetic.

prover has to be familiar not only with the abstract theory on which his proof relies but also with its mechanized counterpart. Like it is the crux of numerical analysis that floating point numbers do not have the nice properties of real numbers, it is the dilemma of automated theorem proving that the mechanization of a logic does not preserve many of its desirable properties. Therefore, a proof certified by a mechanism is actually more difficult than a proof certified by a human. But it is also more reliable.

Let us summarize some of the difficulties that we encountered in the automated as opposed to human certification of trace transformations.

Automated provers work by a set of heuristics. The human who develops the proof is best advised to follow these heuristics. Good heuristics are, of course, those that are naturally followed in many proofs. When the heuristics fail, the human has to document his proof strategy with "proof hints". If a proof is loaded with proof hints, it is probably not tailored very well to the automated prover. (This could indicate a bad proof or a bad prover.) We have spent considerable effort on minimizing and structuring proof hints.

A proof assertion may have many different representations. For instance, all of the formulas below represent the same assertion about a, b, and c:

(a) $a^2+b^2 = c^2$

(b) $a^2+b^2-c^2 = 0$

(c) $c^2-a^2-b^2 = 0$

(d) $aa+bb = cc$

An automated prover may not recognize an assertion in all representations - unless it happens to be an expert on this particular class of assertions. Boyer & Moore's prover, for instance, is not enough of an expert in algebra to treat representations (a), (b), (c), and (d) equivalently. The human has to make sure that the proof uses only representations that the prover can treat as is desired. This can be accomplished by either disciplining the proof or educating the prover, i.e., making it aware of equivalent representations. Education of the prover is a two-edged sword. With too much knowledge, it may spend a long time searching for appropriate facts or even apply at points inappropriate proof rules.

One major concern of automated certification is execution efficiency. The most fundamental efficiency requirement is termination. An inappropriate choice of proof steps may lead to an infinite computation. For instance, many automated provers, like Boyer & Moore's, rewrite equalities only in one direction in order to avoid infinite looping. E.g., with the knowledge of A=B, Boyer & Moore's prover will substitute B for A in proofs, but not vice versa. This has immediate consequences for the implementation of our theory: semantic declarations may be exploited only in one direction. In any particular proof, we

may commute left or commute right, but not both; we may use idempotence to compress traces or expand traces, but not both; we may increase or decrease the parallelism in a trace, but not both.[8]

When solving a programming problem, a programmer has the choice of programming in an existing language, or designing a new language which is particularly suited for the class of problems that he is investigating. A new "special purpose" language may permit him to write more natural programs and may yield more efficient executions. An existing "general purpose" language may grant him more flexibility in reformulating the problem or moving to a different problem class altogether. The same choice presents itself in mechanizing certification. One might use an existing general purpose prover, or one might build a new special purpose prover. In choosing Boyer & Moore's mechanized logic, we have taken the "general purpose" option. We prefer general certification power for the development of our mechanized theory of trace transformations and, in the long run, we do not want to confine ourselves to the language of sorting networks. Boyer & Moore's prover is a suitable and user-friendly tool for the implementation of specialized theories.

5. 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.

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 reduces the burden on the mechanical prover and permits highly automated and reasonably practical verification systems, the price paid is that the deduction system 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. We have chosen Boyer & Moore's logic as our deduc-

[8] It turns out that a transformation from parallel to sequential is more easily certified than a transformation from sequential to parallel as the methodology suggests.

tion system. Other mechanized logics have been used for similar purposes, e.g., PL/CV [2, 6] and LCF [5, 8].

Our experience in mechanical proofs of trace equivalences suggests that a mechanical prover will, in general, follow the clean strategy of an on-paper proof, if it is communicated properly. 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 mechanical 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 meets his needs and is appropriately represented in the mechanized logic. He does not have to be concerned with any aspects of the proof.[9]

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 mechanical 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 J Moore and Bob Boyer who patiently answered our countless questions about their prover. Several discussions with J Moore helped putting our work into perspective.

Appendix: Events of Proof Session

This appendix presents all commands (so-called "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 [4] 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`.

[9] 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 theorems TAU-MAIN is of no concern to him.

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 types 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)`.

   `(USE` ev_1 ... ev_n`)`
   : Enforce the use of axioms or lemmas ev_1 to ev_n. Each ev_i has the form `(name (`v_1 t_1`) ... (`v_n t_n`))`, where `name` is the name of an axiom or lemma to be used, v_i is one of the free variables of `name`, and t_i is a substitution term for v_i.

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))))
```

References

1. Ackerman, W. B. "Data Flow Languages". *Computer 15*, 2 (Feb. 1982), 15-25.

2. Bates, J. L., and Constable, R. L. Proofs as Programs. TR 82-530, Cornell University, 1983.

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

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

5. Cohn, A. J. "The Equivalence of Two Semantic Definitions: A Case Study in LCF". *SIAM Journal of Computing 12* (1983), 267-285.

6. Constable, R. L., Johnson, S. D., and Eichenlaub, C. D.. *An Introduction to the PL/CV2 Programming Logic*. Lecture Notes in Computer Science 135, Springer Verlag, 1982.

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

8. Gordon, M. J. C., Milner, A. J., Wadsworth, C. P. *Edinburgh LCF*. Lecture Notes in Computer Science 78, Springer Verlag, 1979.

9. Huang, C.-H., and Lengauer, C. The Automated Proof of a Trace Transformation for a Bitonic Sort. TR-84-30, Department of Computer Sciences, The University of Texas at Austin, 1984.

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

11. 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.

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

13. Lengauer, C. "On the Role of Automated Theorem Proving in the Compile-Time Derivation of Concurrency". *Journal of Automated Reasoning 1* (1985). To appear. Earlier version: On the Mechanical Transformation of Program Executions to Derive Concurrency. TR-83-20, Department of Computer Sciences, The University of Texas at Austin, Oct., 1983.