A MECHANICALLY CERTIFIED THEOREM ABOUT OPTIMAL CONCURRENCY OF SORTING NETWORKS, AND ITS PROOF [1]

Christian Lengauer and Chua-Huang Huang

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188

TR-85-23 October 1985

Abstract

Our concern is the mechanical certification of transformations of sequential program executions into parallel executions with equivalent semantics. The objective of such transformations is to accelerate the execution of programs. The result reported here is a mechanically certified theorem of optimality. We present a transformation which applies to every program in a particular programming language, the language of sorting networks. This transformation transforms the sequential execution of any sorting network into an execution which is as fast or faster than any other transformation (permitted in our theory) that applies to every sorting network. The theorem is stated formally in a mechanized logic, and its proof as performed by a mechanical theorem prover is described.

[1] This theorem will be presented without proof at the *13th Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13-15, 1986.

1. Introduction

A formal semantics can provide an accurate description of a programming language, and program verification can establish the correctness or incorrectness of a program precisely. But how can we be sure that an alleged proof of correctness of some semantic property actually is a proof? We have to rely on our understanding of the theory in which the alleged proof is carried out and on our unsusceptibility to suggestive but incorrect arguments.

Computer programs have been developed that can check the validity of proofs. The proposition to be proved has to be expressed in the "mechanized" logic in which such a program "reasons". Consequently, we can check proofs of theorems about programming languages and programs by "implementing" the formal semantics of the programming language in a mechanized logic and letting the associated prover check our deductions within that semantics. A (correct) mechanical prover is unsusceptible to suggestive but incorrect deductions. To believe a mechanically checked property, one has to believe that a correct implementation of the mechanical prover has been used, that the formal theorem expresses the desired property, and that no invalid assumptions have entered into the proof. Under these premisses, one need not understand the details of the proof.

Our concern is the mechanical certification of transformations of sequential program executions into parallel executions with equivalent semantics. The objective of such transformations is to accelerate the execution of programs. We call program executions *traces*.

The result reported here is a mechanically certified theorem of *optimality*. We present a trace transformation which applies to every program in a particular programming language and show that this transformation transforms the sequential execution of the program into an execution which is as fast or faster than any other transformation (permitted in our theory) that applies to every program in the language. The language of our choice is sorting networks. We present the formal representation of this theorem in the mechanized logic in which it has been certified, and describe its proof.

2. The Programming Language

Our programming language is a refinement language with the following features:

- The definition of a *refinement* consists of a refinement name with an optional list of formal parameters, separated by a colon from a refinement body. The following are the only three choices of refinement body.

- The *null statement*, `skip`, does nothing.

- The *basic statement* is a statement that is not refined any further. We denote our basic statement `cs(i1,i2)` and call it, following Knuth [9], a *comparator module*. Comparator module `cs(i1,i2)` accesses an array `a[0..n]` of natural numbers. It compares elements `a[i1]` and `a[i2]` 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)`. The comparator module is of imperative nature, i.e., its implementation requires updates.

- 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.

Programs in this language are called *sorting networks* [9].

Example:

The following program represents a network that performs an insertion sort [9]:

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

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

For example, if composition is implemented by execution in sequence, the execution of this program is for a six-element array (n=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)
```

The arrow denotes sequential execution. If we assert unit execution time for comparator modules, this trace has execution time n(n+1)/2, i.e., quadratic in the length of the array.

Comparator modules have a useful property that we can exploit to accelerate the execution of the insertion sort: if they do not access common array elements, their application in commuted order or in parallel will not affect the result of the program. Simple comparator modules whose arguments differ by at least 2 do not access common array elements. Thus, we are allowed to transform the previous sequential trace into the following parallel trace by "ravelling" comparator modules:

```
             /cs(1)\   /cs(2)\   /cs(1)\   /cs(2)\   /cs(1)\
cs(1)→cs(2)→<       >→<       >→< cs(3) >→<       >→<       >→cs(2)→cs(1)
             \cs(3)/   \cs(4)/   \cs(5)/   \cs(4)/   \cs(3)/
```

Angle brackets denote parallel execution. This transformation can be expressed recursively for arrays of any length, and is proved by induction on the length of the array. If we assume instantaneous forks and joins, the parallel trace has execution time 2n−1, i.e., linear in the length of the array.

(End of Example)

We have performed mechanical certifications of individual transformations of three sorting networks [12]: the insertion sort, the odd-even transposition sort, and the bitonic sort. The parallel odd-even transposition sort sorts at double the speed of the parallel insertion sort, but still in linear time. The bitonic sort is a fancier sorting network with an even faster parallel trace: $\log^2$ in the length of the array.

3. The Mechanized Logic

Most interesting programs contain recursions or loops. The most effective and practical trace transformations of such programs are also recursive, and their proofs of correctness require induction. We use the Boyer-Moore induction prover [2]. It employs a mechanized logic that is particularly suitable for reasoning about programs [1]. The logic is functional (i.e., predicates are expressed as functions with a boolean range) and quantifier-free (i.e., free variables are taken as universally quantified). The syntax that the prover accepts looks very much like LISP, but we shall use here a mixture of infix and s-expression prefix notation that we believe is easier to decipher. The prover is designed to prove theorems about recursive functions but is not an expert on language semantics. We have to "teach" it our theory of trace transformations by providing appropriate function definitions and having it certify useful theorems about trace transformations. We call such a collection of functions and theorems the *implementation* of our theory, or the *mechanized* theory. The prover is able to use the function definitions and theorems of the mechanized theory in the certification of further semantic propositions. Details of several versions of our mechanized theory and applications in it can be found in [7, 13].

Our approach differs from the approach of *verification condition generation* [6, 8, 15]. We do not employ a verification condition generator but make the entire semantic theory available to the prover directly. Verification condition generators are built to assist in proofs that programs satisfy specifications but do not support the certification of other theorems about programs and programming languages, e.g., our theorem of optimality. Other mechanized logics in which work similar to ours can be carried out are PL/CV2 [3] and LCF [5].

4. The Representation and Meaning of Traces

We represent a trace of comparator modules in the mechanized logic as a multi-level list. The atomic elements of the list are pairs of numbers[2] or, for simple comparator modules, numbers. Alternate list levels represent sequential execution and parallel execution in turn. For instance, if the top level of the list is executed in sequence, the sequential trace and parallel trace of the previous insertion sort example are represented as

```
 (TAU 5)   =  '(1  2 1  3 2 1  4 3 2 1  5 4 3 2 1)
(TAU~ 5)   =  '(1 2 (3 1) (4 2) (5 3 1) (4 2) (3 1) 2 1)
```

The interpretation of these lists as traces of comparator modules is given by a "weakest precondition generator", a function M-CS which expects as parameters: a list L which represents a trace, a switch FLAG which indicates the modus of execution of the top level of L (FLAG='SEQ for sequential, FLAG='PAR for parallel), and a predicate R which represents a postcondition. M-CS returns a predicate which represents the weakest precondition of L with respect to R [4]. Essentially, interpreting the atomic elements of list L as comparator modules, M-CS composes the appropriate meaning of the sequential or parallel execution of these comparator modules, as prescribed by L. For parallel execution, comparator modules must be independent. Function M-CS has been presented in previous publications [7, 13]. Regarding our theorem of optimality, M-CS is a side issue and we shall not define it here. But we must elaborate on the interpretation of the atomic list elements as comparator modules.

We express the semantics of the comparator module by a function that expects an atomic list element I and a predicate R:

Unspecified Function (CS I R)

We intend (CS I R) to stand for the weakest precondition of comparator module cs(i1,i2), where the pair (i1,i2) is encoded in parameter I, with respect to postcondition R. But instead of defining function (CS I R) we leave it unspecified. We shall not reason about the actual semantics of a trace but only about the *preservation* of this semantics. It turns out that we only need to assert one property of comparator modules. As any language statement, comparator modules must obey the Law of the Excluded Miracle [4]:

Axiom CS-IS-NOT-MIRACLE: (CS I F) = F

F denotes the constant "false".

---

[2]With *atomic* we mean not further decomposable by list accessing operations. Numbers can be extracted from pairs but not by list accessing operations.

We are interested in the execution times of traces. Function (EXEC-TIME FLAG L) computes the execution time of trace L whose modus of execution is determined by FLAG (sequential if FLAG='SEQ, and parallel if FLAG='PAR). In accordance with our trace representation, FLAG alternates for successive levels of L:

Function (EXEC-TIME FLAG L)

```
       =
(IF (NLISTP L)
    (IF L=NIL
        0
        1)
    (IF FLAG='PAR
        (MAX (EXEC-TIME 'SEQ (CAR L))
             (EXEC-TIME 'PAR (CDR L)))
        (PLUS (EXEC-TIME 'PAR (CAR L))
              (EXEC-TIME 'SEQ (CDR L)))))
```

NLISTP is the negation of recognizer LISTP. The value of (IF t t1 t2) is that of t2 if t=F and that of t1 otherwise. Recall that we assume unit execution time of comparator modules, and instantaneous forks and joins for parallel execution.

5. The Transformation of Traces

Trace transformations are justified by certain semantic properties, so-called *semantic relations* [10, 11], that comparator modules may or may not satisfy:

(a) A comparator module that is *idempotent* can be executed once or any number of times consecutively with identical effect.

(b) Two comparator modules that are *commutative* can be executed in any order with identical effect.

(c) Two comparator modules that are *independent* can be executed in parallel and in sequence with identical effect. Independence implies commutativity but, in general, not vice versa.

In the mechanized logic, the idempotence, commutativity, and independence of comparator modules are expressed by the following functions:

Unspecified Function (IDEM I)

Unspecified Function (COM I J)

Unspecified Function (IND I J)

Parameters I and J stand for pairs of numbers (representing comparator modules). Just as we characterize the weakest precondition of comparator modules, we also characterize their semantic relations only in part. First of all, we must identify all semantic relations as predicates:

Axiom IDEM-IS-PREDICATE: (OR (TRUEP (IDEM I)) (FALSEP (IDEM I)))

Axiom COM-IS-PREDICATE: (OR (TRUEP (COM I J)) (FALSEP (COM I J)))

Axiom IND-IS-PREDICATE: (OR (TRUEP (IND I J)) (FALSEP (IND I J)))

TRUEP and FALSEP recognize the truth values T and F, respectively. More importantly, we assert that idempotence and commutativity can be exploited in traces as follows (for a proof see [11], Sect. 5.2):

Axiom `IDEM-ELIMINATES-CS:` `(IDEM I)` $\Rightarrow$ `((CS I (CS I R)) = (CS I R))`

Axiom `COM-SWAPS-CS:` `(COM I J)` $\Rightarrow$ `((CS J (CS I R)) = (CS I (CS J R)))`

Additionally, we assert the following properties of commutativity and independence:

Axiom `IND-IMPLIES-COM:` `(IND I J)` $\Rightarrow$ `(COM I J)`

Axiom `COM-IS-SYMMETRIC:` `(COM I J)` $\Rightarrow$ `(COM J I)`

Axiom `IND-IS-SYMMETRIC:` `(IND I J)` $\Rightarrow$ `(IND J I)`

Comparator modules that are supposed to be executed in parallel must pass the test of independence.[3]

IDEM, COM, and IND relate single comparator modules. There are functions that build on them and relate traces of comparator modules. In this paper, we shall require (IS-IND I L) which establishes the independence of comparator module I with every comparator module in trace L, and (ARE-IND L1 L2) which establishes the mutual independence of comparator modules in trace L1 with comparator modules in trace L2. To establish independence, we need to look only at the sets of comparator modules in the traces, not at the traces' structure. Function (ALL-ATOMS L) returns the set (actually, the bag) of comparator modules of trace L. Arguments of IS-IND and ARE-IND must be processed by ALL-ATOMS. Consult the appendix for formal definitions of these functions.

6. The Space of Transformations of a Sequential Trace

In order to establish the optimality of a transformation of some sequential trace L1, we establish the space of legal transformations L2 of L1 and then prove the optimality of one of these transformations. The legal transformations of L1 are recognized by a function (TRANSFORMABLE L1 L2). This function expects a sequential trace L1 and a feasible trace L2 of comparator modules and establishes whether trace L2 can be derived by correct exploitations of idempotence, commutativity, and independence of individual comparator modules from trace L1. A *sequential* trace may not contain parallel commands, and a *feasible* trace may not contain parallel commands with dependent elements. If L1 and L2 satisfy (TRANSFORMABLE L1 L2), L2 must be obtained from L1 by adding, deleting, commuting, or parallel merging comparator modules in L1. We need not consider commutations other than those implied by independence. The only situation where two comparator modules are commutative but not independent is when both are identical. The according commutation is the identity transformation. For example, with sequential trace '(1 3 5 1 2 4 6)[4], the following traces are accepted by TRANSFORMABLE:

(1) '(1 3 5 2 4 6)
{one occurrence of 1 is deleted because it is idempotent and can be commuted with 3 and 5}

(2) '((1 3 5) 3 1 (2 4) 6)
{one occurrence of 3 is added and commuted, and two parallel commands '(1 3 5) and '(2 4) are generated}

(3) '((1 3 5) ((1 2) 4 6))
{the second top-level component contains a sequential subtrace '(1 2) which is parallel to 4 and 6}

but the following traces are not:

[3]This is part of the definition of M-CS.

[4]For clarity of exposition, example traces in this paper contain only simple comparator modules.

(4) `'(3 5 2 4 6)` {1 is missing}

(5) `'(5 (1 3 2) 4 6)` {would be accepted by TRANSFORMABLE, but is not a feasible trace}

(6) `'((1 5) 2 3 (4 6))` {2 cannot be commuted with 3}

In order to prevent faulty applications, as in case (5), we need to check that the arguments of TRANSFORMABLE meet the requirements imposed on them. Function SEQUENTIAL recognizes the sequentiality of a trace. It checks that the elements of the list that represents the trace are proper representations of comparator modules:

```
Function   (SEQUENTIAL L)
               =
           (IF (NLISTP L)
               L=NIL
               (AND (COMP-MOD (CAR L))
                    (SEQUENTIAL (CDR L))))
```

For simple comparator networks, recognizer COMP-MOD is NUMBERP, the recognizer for numbers; for general comparator networks, COMP-MOD is PAIRP, the recognizer for pairs of numbers.

Function FEASIBLE recognizes the feasibility of a trace. A feasible trace must consist of comparator modules, and elements of parallel commands in the trace must be independent:

```
Function   (FEASIBLE FLAG L)
               =
           (IF (NLISTP L)
               L=NIL
               (IF FLAG='PAR
                   (IF (ARE-IND (ALL-ATOMS (CAR L))
                                (ALL-ATOMS (CDR L)))
                       (IF (NLISTP (CAR L))
                           (AND (COMP-MOD (CAR L))
                                (FEASIBLE 'PAR (CDR L)))
                           (AND (FEASIBLE 'SEQ (CAR L))
                                (FEASIBLE 'PAR (CDR L))))
                       F)
                   (IF FLAG='SEQ
                       (IF (NLISTP (CAR L))
                           (AND (COMP-MOD (CAR L))
                                (FEASIBLE 'SEQ (CDR L)))
                           (AND (FEASIBLE 'PAR (CAR L))
                                (FEASIBLE 'SEQ (CDR L))))
                        F)))
```

The concept of transformability rests on the notion of reachability. A comparator module I is *reachable* in a trace L if it occurs at least once in L and the first occurrence can be commuted with every comparator module prior to it in L. Function REACHABLE recognizes reachability:

Function
```
(REACHABLE I L)
    =
(IF (NLISTP L)
    F
    (IF (LIST I) = (ALL-ATOMS (CAR L))
        T
        (IF (MEMBER I (ALL-ATOMS (CAR L)))
            (REACHABLE I (CAR L))
            (AND (IS-IND I (ALL-ATOMS (CAR L)))
                 (REACHABLE I (CDR L))))))
```

TRANSFORMABLE uses a helping function (REMOVE I L) that removes the first occurrence of element I from list L and, if I is the single element of a list, i.e., (I), or ((I)), or (((I))), etc., it removes also the resulting empty lists (see appendix). Here is, finally, the formal definition of TRANSFORMABLE, followed by an explanation:

Function
```
     (TRANSFORMABLE L1 L2)
         =
(1)  (IF (NLISTP L1)
(2)      L2=NIL
(3)      (IF (REACHABLE (CAR L1) L2)
(4)          (IF (IDEM (CAR L1))
(5)              (IF (REACHABLE (CAR L1) (CDR L1))
(6)                  (TRANSFORMABLE (CDR L1) L2)
(7)                  (IF (REACHABLE (CAR L1) (REMOVE (CAR L1) L2))
(8)                      (TRANSFORMABLE L1 (REMOVE (CAR L1) L2))
(9)                      (TRANSFORMABLE (CDR L1) (REMOVE (CAR L1) L2))))
(10)             (TRANSFORMABLE (CDR L1) (REMOVE (CAR L1) L2)))
(11)         F))
```

L2 is recognized as transformable from L1 in one of the following cases: first, if L1 is the null trace L2 must be null, too (lines 1 and 2 of the function body). If L1 is not null, then the first comparator module of L1, (CAR L1),[5] must be reachable in L2 (line 3); otherwise, it is either not part of L2 or it is commuted with a dependent comparator module and, therefore, L2 cannot be legally derived from L1. So, let us assume the first comparator module of L1 is reachable in L2. Then, if it is an idempotent comparator module (line 4), it might be eliminated or duplicated to derive L2. To be eliminated from L1, (CAR L1) must be reachable in (CDR L1) (line 5). Then, transformability of (CDR L1) into L2 is tantamount to the elimination of (CAR L1) from L1 (line 6). For the purpose of transformability, duplication in L1 is the same as elimination from L2. To be eliminated from L2, (CAR L1) must be reachable in L2 without the first occurrence of (CAR L1) (line 7). Then, transformability of L1 into L2 without the first occurrence of (CAR L1) is tantamount to the elimination of (CAR L1) to L2 (line 8). Otherwise, transformability of (CDR L1) into L2 without the first occurrence of (CAR L1) is tantamount to not exploiting the idempotence of (CAR L1) in the derivation of L2 (line 9). Idempotence is also not exploited if (CAR L1) is not even idempotent (line 10).

(TRANSFORMABLE L1 L2) recognizes all legal exploitations of idempotence and commutativity in trace L1. Independence is not addressed in TRANSFORMABLE, since TRANSFORMABLE expects a sequential trace for L1 and a feasible trace for L2. Independence is not an issue in sequential traces and is checked in function FEASIBLE. We have mechanically certified that (TRANSFORMABLE L1 L2) guarantees the semantic identity (i.e., the identity of the weakest preconditions) of traces L1 and L2.

[5]Remember that L1 is sequential.

7. The Optimal Transformation

We define a trace transformation (TRANSFORM L) which takes a sequential trace L of comparator modules and yields another trace which may or may not contain parallelism. (TRANSFORM L) is L "ravelled" into concurrency, if possible. TRANSFORM transforms a single-level sequential trace into a two-level trace that contains only parallel commands. A parallel command with only one member means sequential execution. The main work is performed by a helping function (RAVEL I L) which adds a comparator module I to a two-level trace L. We define RAVEL first and then explain it:

```
Function   (RAVEL I L)
               =
     (1)   (IF (NLISTP L)
     (2)       (LIST (LIST I))
     (3)       (IF (AND (IDEM I)
     (4)                (REACHABLE I L))
     (5)           L
     (6)           (IF (IS-IND I (ALL-ATOMS (CAR L)))
     (7)               (IF (OR (NLISTP (CDR L))
     (8)                       (NOT (IS-IND I (ALL-ATOMS (CADR L)))))
     (9)                 (CONS (CONS I (CAR L)) (CDR L))
     (10)                  (CONS (CAR L) (RAVEL I (CDR L))))
     (11)                (CONS (LIST I) L))))
```

If L is empty, the result of (RAVEL I L) is ((I)) (lines 1 and 2 of the function body). For non-empty L, if I is idempotent and occurs in L with only comparator modules independent from I to its left (lines 3 and 4), then I is discarded and the result is L (line 5). Otherwise, if I can be commuted into L (line 6), it is (line 10), until L is exhausted or one comparator module in the parallel command to I's right depends on I (lines 7 and 8), and then I is merged with the parallel command to its left[6] (line 9). If I cannot be commuted into L, then (RAVEL I L) adds a single-member parallel command (I) to the front of L (line 11). Here are some applications of RAVEL:

(1) (RAVEL 1 NIL) = '((1))
{NIL represents the empty trace}

(2) (RAVEL 3 '((1 3 5) (2 4 6))) = '((1 3 5) (2 4 6))
{3 is idempotent, independent with 1, and occurs in '((1 3 5) (2 4 6)); therefore, it is discarded}

(3) (RAVEL 8 '((1 3 5) (2 4 6))) = '((1 3 5) (8 2 4 6))
{8 commutes to the end of the trace and then merges with '(2 4 6)}

(4) (RAVEL 7 '((1 3 5) (2 4 6))) = '((7 1 3 5) (2 4 6))
{7 is independent with parallel command '(1 3 5), but not with '(2 4 6)}

(5) (RAVEL 2 '((1 3 5) (2 4 6))) = '((2) (1 3 5) (2 4 6))
{2 is not independent with 1 or 3}

Function (TRANSFORM L) ravels trace L element after element:

[6] Remember that any two comparator modules that can be commuted can be merged, since our criterion for either is independence.

Function

```
(TRANSFORM L)
      =
(IF (NLISTP L)
    NIL
    (RAVEL (CAR L) (TRANSFORM (CDR L)))))
```

Examples of TRANSFORM are:

(1) `(TRANSFORM '(5 4 3 2 1)) = '((5) (4) (3) (2) (1))` {sequential execution}

(2) `(TRANSFORM '(1 3 5 2 4 6)) = '((1 3 5) (2 4 6))`

(3) `(TRANSFORM '(1 2 4 6 4 2 1)) = '((1) (2) (6 4 1))`

(4) `(TRANSFORM '(1  2 1  3 2 1  4 3 2 1  5 4 3 2 1))` {insertion sort: `(TAU 5)`}
`= '((1) (2) (1 3) (2 4) (1 3 5) (2 4) (1 3) (2) (1))`

The following theorem states that `(TRANSFORM L)` is a feasible trace and a legal transformation of sequential trace L:

Theorem TRANSFORM-IS-FEASIBLE-AND-TRANSFORMABLE:

```
   (SEQUENTIAL L)
⇒ (AND (FEASIBLE 'SEQ (TRANSFORM L))
       (TRANSFORMABLE L (TRANSFORM L)))
```

It has been mechanically certified, but we shall omit the proof.

8. The Proof of Optimality

Given that TRANSFORM defines a legal transformation of its sequential argument, the theorem of optimality states that this transformation yields an execution that is faster than or as fast as any other legal transformation:

Theorem OPTIMALITY:

```
   (AND (SEQUENTIAL L1)
        (FEASIBLE FLAG L2)
        (TRANSFORMABLE L1 L2) )
⇒ ( (EXEC-TIME 'SEQ (TRANSFORM L1)) ≤ (EXEC-TIME FLAG L2) )
```

The main idea of the proof is to define a "normal form" of traces and show that this normal form optimizes the execution time of any trace. Then we can reduce the space of transformations L2 that we must consider to normal traces. It turns out that the reduced space contains only optimal transformations of L1, i.e., the execution time of `(TRANSFORM L1)` equals the execution times of all normal transformations of L1.

We define the normalization of traces in two steps:

(1) A function `(COMPACT FLAG L)` first puts feasible trace L into a compact form which preserves feasibility, transformability, and execution time. A feasible trace of comparator modules is *compact* if its top level is executed in sequence and the trace has exactly two levels. In a compact trace, every comparator module is part of a parallel command and comparator modules are not sequentially composed within parallel commands.

(2) A function (NORMALIZE L) then puts compact trace L into its normal form which preserves compactness and transformability, but optimizes execution time. A compact trace is *normal* if, for any two adjacent parallel commands, there exists no comparator module in the first parallel command that is independent with all comparator modules in the second parallel command or, if it is idempotent, is identical to one of the comparator modules in the second parallel command. Also, parallel commands may not contain duplicate idempotent elements. In a normal trace, no comparator module can be commuted any further to the right or deleted.

Let us proceed in sequence and first define COMPACT. COMPACT uses a function APPEND-ALL which takes two lists whose elements are lists. (APPEND-ALL L1 L2) appends the lists in L2 to the respective lists in L1 (see appendix):

```
Function  (COMPACT FLAG L)
              =
          (IF (NLISTP L)
              (IF L=NIL
                  NIL
                  (LIST (LIST L)))
              (IF FLAG='PAR
                  (APPEND-ALL (COMPACT 'SEQ (CAR L))
                              (COMPACT 'PAR (CDR L)))
                  (APPEND (COMPACT 'PAR (CAR L))
                          (COMPACT 'SEQ (CDR L)))))
```

Examples of COMPACT are:

(1) (COMPACT 'SEQ '((1 3 5) 2 4 (5 7))) = '((1 3 5) (2) (4) (5 7))
{2 and 4 are transformed to two single-member parallel commands; both traces have execution time 4}

(2) (COMPACT 'PAR ((1 2 3) (5 6))) = '((1 5) (2 6) (3))
{trace <1→2→3 5→6> is transformed to <1 5>→<2 6>→<3>; both have execution time 3}

With a function (COMPACTP L), which recognizes the compactness of trace L and is not presented in this paper, we can prove the following properties of COMPACT, provided the premisses of theorem OPTIMALITY are given:

Lemma COMPACT-IS-COMPACTP: {COMPACT returns a compact trace}

(COMPACTP (COMPACT FLAG L2))

Lemma COMPACT-PRESERVES-FEASIBLE: {COMPACT preserves feasibility}

(FEASIBLE 'SEQ (COMPACT FLAG L2))

Lemma COMPACT-PRESERVES-TRANSFORMABLE: {COMPACT preserves transformability}

(TRANSFORMABLE L1 (COMPACT FLAG L2))

Lemma COMPACT-PRESERVES-EXEC-TIME: {COMPACT preserves execution time}

(EXEC-TIME FLAG L2) = (EXEC-TIME 'SEQ (COMPACT FLAG L2))

Next, we define NORMALIZE. NORMALIZE uses a helping function TRANSFORM-LIST that generalizes function TRANSFORM. Whereas (TRANSFORM L1) ravels the elements of trace L1 into the empty trace, (TRANSFORM-LIST L1 L2) ravels the elements of L1 into a given trace L2 (see appendix), i.e., (TRANSFORM L1) = (TRANSFORM-LIST L1 NIL):

Function
```
(NORMALIZE L)
    =
(IF (NLISTP L)
    L
    (TRANSFORM-LIST (CAR L)
                    (NORMALIZE (CDR L))))
```

Examples of NORMALIZE are:

(a) `(NORMALIZE '((1 3 5 7) (2 4 6) (1 3))) = '((1 3 5) (7 2 4) (6 1 3))`
{normalization does not improve the execution time}

(b) `(NORMALIZE '((1 3) (2 5) (4 6))) = '((1 3 5) (2 4 6))`
{normalization reduces the execution time by 1}

With a function (NORMAL L), which recognizes the normality of compact trace L and is not presented in this paper, we can prove the following properties of NORMALIZE, provided the premisses of theorem OPTIMALITY are given, and L2 is compact:

Lemma NORMALIZE-IS-NORMAL: {the result of NORMALIZE is in normal form}

`(NORMAL (NORMALIZE L2))`

Lemma NORMALIZE-PRESERVES-COMPACTP: {NORMALIZE preserves compactness}

`(COMPACTP (NORMALIZE L2))`

Lemma NORMALIZE-PRESERVES-TRANSFORMABLE: {NORMALIZE preserves transformability}

`(TRANSFORMABLE L1 (NORMALIZE L2))`

Lemma NORMALIZE-IS-OPTIMAL: {NORMALIZE optimizes execution time}

(EXEC-TIME 'SEQ (NORMALIZE L2)) $\leq$ (EXEC-TIME 'SEQ L2)

We shall not prove any of these eight lemmas here, but sketch the idea of the proof of the most interesting lemma, NORMALIZE-IS-OPTIMAL. It deduces inductively from the definition of NORMALIZE that a normal trace cannot be sped up any further by legal transformations, and is based on the fact that ravelling a parallel command into a compact trace must either maintain the execution time of that trace or increase it by 1. Each parallel command that NORMALIZE ravels recursively into the already normalized tail of its compact argument is either absorbed, decreasing execution time by 1, or at least one of its comparator modules forms one new parallel command tacked on to the front, preserving execution time.

This completes our description of the normalization of traces. Actually, our optimal transformation TRANSFORM simply normalizes its argument: (TRANSFORM L) = (NORMALIZE (COMPACT 'SEQ L)). TRANSFORM is simpler than NORMALIZE in the sense that (TRANSFORM L) expects a sequential L, whereas (NORMALIZE (COMPACT L)) deals with any feasible L.

The following theorem is our claim of optimality for a reduced space of transformations L2, namely, normal traces. All elements of the reduced space are optimal, i.e., as fast as (TRANSFORM L1):

Lemma NORMAL-OPTIMALITY:

```
    (AND (SEQUENTIAL L1)
         (COMPACTP L2)
         (NORMAL L2)
         (TRANSFORMABLE L1 L2) )
 ⇒ ( (EXEC-TIME 'SEQ (TRANSFORM L1)) = (EXEC-TIME 'SEQ L2) )
```

In fact, we can prove something even stronger. For all practical purposes, each transformation L2 of L1 normalizes to the same trace: (TRANSFORM L1). The proof of NORMAL-OPTIMALITY, essentially, rewrites an arbitrary normal transformation L2 of L1 into (TRANSFORM L1). That does not mean, however, that (TRANSFORM L1) is the unique optimal transformation of L1!

Formally, OPTIMALITY is proved from NORMAL-OPTIMALITY by the following sequence of steps:

```
  (EXEC-TIME 'SEQ (TRANSFORM L1))
  {COMPACT-IS-COMPACTP, COMPACT-PRESERVES-TRANSFORMABLE,
   NORMALIZE-IS-NORMAL, NORMALIZE-PRESERVES-COMPACTP,
   NORMALIZE-PRESERVES-TRANSFORMABLE, and NORMAL-OPTIMALITY}
= (EXEC-TIME 'SEQ (NORMALIZE (COMPACT FLAG L2)))
  {COMPACT-IS-COMPACTP and NORMALIZE-IS-OPTIMAL}
≤ (EXEC-TIME 'SEQ (COMPACT FLAG L2))
  {COMPACT-PRESERVES-EXEC-TIME}
= (EXEC-TIME FLAG L2)
```

This concludes our description of the proof of optimality.

9. On Mechanical Certification

Published proofs of any appreciable depth will always resort to "handwaving", i.e., slip into informality. However, while conventional proofs stop there, we have continued the formalization of our proof to the point where it has been accepted by a mechanical prover that believes only the most primitive axioms[7] (plus the axiomatic assumptions that we have provided in addition, all of which are reported here). With the authority of a mechanical prover on our side, we have been more precise in the statement of the theorem and more superficial in the description of its proof than we would have been had we performed a conventional "semi-formal" proof. The full extent of our mechanical proof is documented in a log of the proof session; it is a good deal more involved than a conventional proof would have been. But, if you believe the soundness of the theorem proving program, it is very easy to catch potential cheating when inspecting a mechanical proof: just search the proof log for additional axioms that you have not been told about.

10. Conclusions

We look at TRANSFORM as a mapping or *description* of traces, not as an algorithm or *prescription* for the derivation of traces. While the mapping is perfectly well-defined for unbounded traces, an execution of TRANSFORM for an unbounded trace would never terminate.

TRANSFORM yields the fastest executions that can be obtained by exploiting the afore-mentioned semantic relations of individual comparator modules, the basic building blocks that make up every sorting network. It may be possible to improve the execution of a particular network further by a transformation that exploits semantic relations of compositions of comparator modules specific to that one network.

To apply TRANSFORM to a particular sorting network, we must check the conditions under which comparator modules are idempotent, commutative, or independent: with the refinement of cs in Sect. 4,

[7] The Boyer-Moore logic is built on the axioms of mathematical induction and Peano, recursive function theory, propositional logic, and equality [1].

every comparator module is idempotent and comparator modules with disjoint arguments are independent (and therefore commutative). Thus, all we have to do is to replace the test of idempotence IDEM in function TRANSFORM by vacuous tests T, and the tests of independence IND by tests of non-overlap of pairs.[8]

The transformations of the sequential traces of the insertion, odd-even transposition, and bitonic sort alluded to in Sect. 2 differ slightly from what TRANSFORM would yield, but have the same execution time. We have not certified this conjecture mechanically, but could do so with the present implementation of our theory. For a proof that the three transformations cannot be improved individually, we would require notions of transformability that include also network-specific transformations.

Let us summarize again the properties of our programming language which enter into our result:

(1) The language consists of the null statement, one or more basic statements, composition, and recursive refinement.

(2) The basic statements are of imperative nature and are expressed each by a name and a list of arguments. We use one basic statement, the comparator module, name it cs, and provide two natural numbers as arguments (or one natural number for the simple comparator module).

(3) Each basic statement obeys the Law of the Excluded Miracle and takes unit execution time.

(4) The conditions under which basic statements satisfy the various semantic relations are phrased solely in terms of their arguments (for comparator modules it is the true condition for idempotence, and the condition of non-overlap for commutativity and independence).

(5) All interesting commutativity is implied by independence.

Our result can be applied to any programming language with these characteristics.

Acknowledgements

Thanks to Ernie Cohen for several helpful discussions in the early stages of the proof. The AI Lab of our department provided computing time for the mechanical proof. Partial support for this research was provided by the Lockheed Corporation, Grant No. 26-7603-35.

Appendix

Function
```
(IS-IND I L)
     =
(IF (NLISTP L)
    T
    (AND (IND I (CAR L))
         (IS-IND I (CDR L))))
```

[8]Two pairs do not *overlap* if all of their elements are mutually distinct. The representation of pairs and their non-overlap in the Boyer-Moore logic is described in [14].

Function
```
(ARE-IND L1 L2)
    =
(IF (NLISTP L1)
    T
    (AND (IS-IND (CAR L1) L2)
         (ARE-IND (CDR L1) L2)))
```

Function
```
(ALL-ATOMS L)
    =
(IF (NLISTP L)
    (IF L=NIL
        NIL
        (LIST L))
    (APPEND (ALL-ATOMS (CAR L))
            (ALL-ATOMS (CDR L))))
```

Function
```
(REMOVE I L)
    =
(IF (NLISTP L)
    NIL
    (IF ((LIST I) = (ALL-ATOMS (CAR L)))
        (CDR L)
        (IF (MEMBER I (ALL-ATOMS (CAR L)))
            (CONS (REMOVE I (CAR L)) (CDR L))
            (CONS (CAR L) (REMOVE I (CDR L))))))
```

Function
```
(APPEND-ALL L1 L2)
    =
(IF (NLISTP L1)
    L2
    (IF (NLISTP L2)
        L1
        (CONS (APPEND (CAR L1) (CAR L2))
              (APPEND-ALL (CDR L1) (CDR L2)))))
```

Function
```
(TRANSFORM-LIST L1 L2)
    =
(IF (NLISTP L1)
    L2
    (RAVEL (CAR L1)
           (TRANSFORM-LIST (CDR L1) L2)))
```

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. 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.

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

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

6. Good, D. I. Mechanical Proofs about Computer Programs. In *Mathematical Logic and Programming Languages,* C. A. R. Hoare and J. S. Sheperdson, Eds., Series in Computer Science, Prentice-Hall Int., 1985, pp. 55-75.

7. 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, Oct., 1984.

8. Igarashi, S., London, R. L., and Luckham, D. C. "Automatic Program Verification I: A Logical Basis and its Implementation". *Acta Informatica 4* (1975), 145-182.

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

10. 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.

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

12. 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.

13. Lengauer, C. "On the Role of Automated Theorem Proving in the Compile-Time Derivation of Concurrency". *Journal of Automated Reasoning 1,* 1 (1985), 75-101.

14. Lengauer, C., and Huang, C.-H. Automated Deduction in Programming Language Semantics: The Mechanical Certification of Program Transformations to Derive Concurrency. TR-85-04, Department of Computer Sciences, The University of Texas at Austin, Jan., 1985.

15. Polak, W. *Compiler Specification and Verification.* Lecture Notes in Computer Science 124, Springer-Verlag, 1981.