Journal of Automated Reasoning 1 (1985) 75–101. 0168–7433/85/0011–0075$04.05

On the Role of Automated Theorem Proving in the Compile-Time Derivation of Concurrency

CHRISTIAN LENGAUER
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188, U.S.A. (ARPA: cs.lengauer@utexas-20)

(Received 10 February 1984)

Abstract. A recent trend in program development is to derive correct implementations from program specifications by the application of a formal calculus, a 'programming methodology'. The application of formal rules lends itself to automation. We investigate the automation of one part of a methodology for programming with concurrency. In this methodology, concurrency is derived by transforming the sequential execution of a program into an equivalent concurrent execution on the basis of formal transformation rules. Such rules can be interpreted as theorems of semantic equivalences. The mechanical certification of these theorems would significantly enhance the reliability of the methodology. The following is an initial exploration of this problem applied to a certain class of programs: sorting networks. We present an implementation of a part of the underlying semantic theory in Boyer and Moore's mechanized logic, and report on the mechanical proof of a transformation that derives concurrency for an insertion sort.

Key words: Automated program synthesis and validation, concurrency, program transformation.

1. Introduction

In the last decade, few 'buzzwords' have gained as much popularity in computing as *structured programming* [6]. It refers to a disciplined approach to program development by a set of informal derivation rules (stepwise refinement, divide-and-conquer, case analysis, data abstraction, etc.). The application of such rules is encouraged by suitable programming constructs in 'structured' programming languages. Structured programming (a more appropriate term is 'disciplined' programming) aims to provide guidance and diminish the need for intuition during the process of programming.

The reliability of disciplined programming methods is greatly enhanced when they can be applied on a formal basis. *Program verification* [8, 12] refers to formal methods of proving properties of programs, most importantly, that the program conforms to its specifications. Program verification provides an exact description of programs and thereby of their correctness or incorrectness. But it does not support correct program development.

Programming methodology [7, 10, 11] is a new area of research that attempts to merge structured programming and program verification. A programming methodology is a set of formal program development rules, a 'programming calculus', that can be applied to

transform the specification of a program into a correct implementation. It replaces some of the programmers' intuition by formal guidance.

Unfortunately, we cannot trust programmers' formal program derivations any more than their informal program derivations. Formal derivation rules are often difficult to apply and then just as prone to error as informal derivation rules. However, formal derivation rules can be made precise enough to be mechanically certified. And we have considerably more trust in a mechanical than a human certification. We are therefore interested in a connection between programming methodology and *automated theorem proving* [5], the mechanical application of formal logics.

Concurrency is the execution of several program parts in parallel. Rigor in program development is especially desirable in the presence of concurrency. If the input data are known, the outcome of an execution that performs (deterministic) operations in some defined sequence is totally predictable. But the outcome of an execution that performs operations concurrently depends on the unpredictable order of its concurrent activities. Such dependencies may be buried deep inside the program's operations, or may even be determined by the architecture of the machine on which the program is executed. Simple inspection of the program text cannot reveal them. In addition, the number of possible orders explodes exponentially with the number and length of concurrent activities. To describe the properties of a concurrent execution, a formal treatment is absolutely essential. The verification of programs with concurrency is now quite well developed [18, 23, 25]. However, programming methodologies are only just emerging.

The following section deals with one such methodology [19]. Section 3 briefly discusses our approach to the mechanical support of (a part of) that methodology, and Section 4 provides the technical details. This description of our work is aimed at the automated reasoning community. See [22] for a less technical and more recent assessment of our approach to the automation of the static derivation of concurrency from the viewpoint of formal semantics and verification.

2. A Methodology for Programming with Concurrency

Our goal is to mechanize parts of a particular methodology for the derivation of concurrency in programs [19]. This section describes that methodology.

Two different motives may lead to the application of concurrency:

(1) The desire for a specific program *behavior*. For instance, one might wish to run an experiment which involves certain processes executed by designated processors that communicate and synchronize with each other in some fashion. Such applications are to ensure the correct functioning of some machine configuration with a specific concurrency structure. Examples are distributed or operating systems.

(2) The desire for fast program *results*. For instance, one might wish to execute a numerical or data processing algorithm with concurrency in order to obtain a result faster. Such applications do not refer to a specific machine configuration or concurrency structure, but only to some relation of input and output values. Examples are numerical and sorting algorithms.

The programming methodology described here takes the second approach: concurrency is viewed as a tool for accelerating the acquisition of results, not as a basic characteristic of a program. Consequently, concurrency will not be part of the problem specification, but will be derived after the development of the program. We would like to certify this derivation mechanically.

The specification of a programming problem has three parts:

(a) the input constraints under which the program shall operate,
(b) the results which the program is supposed to achieve, and
(c) the time limit imposed on the program's execution.

The program development then proceeds along the following lines:

(1) Perform a formal stepwise refinement of a program that achieves the desired result under the given input constraints. The program does not address the question of execution order. It may not require its operations to be applied in total order, but an easy sequential execution can, at this point, serve as a first execution time estimate.
(2) Declare simple relations between program components, so-called 'semantic relations', that allow relaxations in sequencing, e.g., concurrency. Do so until the execution time of the program satisfies the specified time limit.

A refinement is, for instance, `C1;C2`. This refinement says that program component `C2` is applied to the results of program component `C1`. The semicolon denotes 'application'. It may be implemented by executing `C1` and then `C2`, but need not be in all cases.

Semantic relations are, for instance, the commutativity of the components `C1` and `C2` (written `C1&C2`), and the independence of `C1` and `C2` (written `C1||C2`). `C1` and `C2` are commutative, i.e., `C1&C2` may be declared if the execution of `C1` and then `C2` has the same effect as the execution of `C2` and then `C1`. If `C1&C2` is declared, `C1;C2` may also be implemented by executing `C2` and then `C1`. `C1` and `C2` are independent, i.e., `C1||C2` may be declared if the execution of `C1` and `C2` in parallel has the same effect as their execution in order. If `C1||C2` is declared, `C1;C2` may also be implemented by executing `C1` and `C2` in parallel. Independence implies commutativity.

Let us illustrate the use of this methodology with an example.

EXAMPLE: Sorting. The problem is to sort an array `a[0..n]` of real numbers into ascending order in time $O(n)$. Our refinement is an insertion sort adapted from Knuth [16]:

```
        sort(n):  for i:=1 to n do S(i) od
        S(0):     skip
(i>0)   S(i):     cs(i); S(i-1)
        |i-j|>1  ⇒  cs(i) || cs(j)
```

Statement `cs(i)` represents a restricted version of Knuth's 'comparator module' [16]: `cs(i)`compares adjacent array elements `a[i-1]` and `a[i]` and, if necessary, swaps them

into order. Knuth's comparator module can deal with not only adjacent but any two array elements. Knuth calls programs whose primitive components are comparator modules *sorting networks*. Let us call our restricted version *simple sorting networks*.

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. (For the underlying theory see [20].)

For, say, a six-element array (`n=5`), the refinement has the following sequential execution, if we interpret iteration (denoted by '**for**') and composition (denoted by ';') as execution in order (denoted by '→') and expand components `S(i)` 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. Remember that independence implies commutativity. We commute comparator modules in `tau(5)` left, and then merge adjacent modules whose indices differ by `2` into a parallel command (denoted by angle brackets):

```
tau~(5) =
cs(1)→cs(2)→⟨cs(1), cs(3)⟩→⟨cs(2), cs(4)⟩→⟨cs(1), cs(3), cs(5)⟩→⟨cs(2), cs(4)⟩→⟨cs(1), cs(3)⟩→cs(2)→cs(1)
```

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. We 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. (*End of Example.*)

To declare semantic relations for some refinement, one does not need to understand the refinement as a whole. A local understanding of the components appearing in the declared relation is sufficient. Note also the simplicity of concurrency: there is no need for synchronization of concurrent components other than at the point of termination. Most semantic declarations come easily to mind and have a simple proof.

But the foremost benefit of this approach to the derivation of fast programs is that the more important and better understood question of program refinement is resolved before the less important and more complex question of concurrency arises. Concurrency is later added in isolated steps (by invoking semantic relations) without changing the approved meaning of the program. Thus the development of programs with concurrency 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 with ';' and '**for**'.

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. Traces are composed with '→' and '< >'.

We call Stage 1 the *refinement calculus* and Stage 2 the *trace calculus*. Either of the two stages has the potential for automation. Automation of Stage 1 would yield a mechanical system for program refinement. Research along these lines is under way elsewhere [2, 24]. Automation of Stage 2 would yield a very powerful optimizing compiler (since we view concurrency as optimization). Early work in this area [17] has been without a formal semantic basis. At that time, formal semantics was in its infancy. Our interest is the mechanical support of Stage 2 on a formal semantic basis.

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. Our approach is 'referentially opaque', i.e., permits the re-assignment of variables and, consequently, requires a more complicated data flow analysis. We have to explicitly declare and subsequently exploit data independencies (in our fomalism, semantic relations).

The vast majority of software that exists today and is currently being produced is referentially opaque. The vast majority of today's machine architectures support the referentially opaque programming style. While we must strive for new programming styles and machine architectures, we must also continue to increase our understanding of the present technology.

3. On the Mechanical Support of Trace Transformations

We shall focus on the mechanical support of the trace calculus. (For initial ideas see Section 6.2.1 of [21].) Let us consider the previous example **sort(n)**. We derived a parallel trace **tau⁻(n)** that satisfies the execution time limit from sequential trace **tau(n)** that is suggested by the refinement of **sort(n)** and does not satisfy the execution time limit. The semantic equivalence of **tau(n)** and **tau⁻(n)** ensures that the meaning of the program is preserved. The equivalence is proved by a recursive application of a sequence of trace transformations that exploit no semantic relations other than those declared for **sort(n)**. Although such transformations are in many cases, as in this one, quite simply described in informal English (see Section 2), their formal application is extremely tedious (see

Section 5.4 of [21].) We do not want to rely on the informal description but would like some mechanical aid in the formal application.

We might be tempted to view a trace transformation as an algorithm. For our example we would need an algorithm, say, **trans(n)** that transforms **tau(n)** into **tau⁻(n)** by appropriately commuting and merging **tau**'s comparator modules. The algorithm we propose uses commands:

commute x left to identify the next trace element of form **x** to the right of the current cursor position and commute it with its left neighbor, and

merge x left to identify the next trace element of form **x** to the right of the current cursor position and merge it and its left neighbor into a parallel command.

Let us assume that, at the start of any transformation, the cursor is placed at the left end, i.e., the head of the trace to be transformed. For the following algorithm, **trans(n)**, the cursor is initially placed at the head of trace **tau(n)**:

```
trans(n): for i:=3 to n
            do for j:=0 to i-3
                 do for k:=1 to i-3-j
                      do commute cs(i-j) left od;
                    merge cs(i-j) left
                 od
            od
```

trans(n) derives **tau⁻(n)** in one left-to-right pass of right-to-left commutations and merges in cubic time ($O(n)$ for every nested **for**). For **n<3**, **tau(n)=tau⁻(n)** and **trans(n)** does nothing.

This approach has the problem that if the refinement contains a recursion (or repetition), as **sort(n)** does, the transformation algorithm refers to the depth of the recursion: for fixed input **n**, **trans(n)** will make **sort(n)** linear, but not **sort(n+1)**. Traces of a recursive refinement are unbounded, and an algorithm like **trans(n)** cannot completely transform an unbounded trace in finite time.

A better approach is to treat trace transformations as theorems, not algorithms. In particular, recursive transformations are inductive theorems. For our transformation of **sort(n)**, the theorem is:

TAU.MAIN: For all **n>0**, **tau⁻(n)** ≡ **tau(n)**

where '≡' denotes semantic equivalence. The proof essentially rewrites one side of the equation into the other. Because it uses induction (on **n**), it can deal with unbounded traces in finite time.

With this view, the automation of trace transformations constitutes a system that, following a set of heuristics and, most likely, with the help of some interactive dialogue, identifies valid and useful transformation theorems. Our present approach is to split this task into two parts:

(a) the identification and (recursive) formulation of trace transformations, and
(b) the proof of the corresponding (inductive) theorems,

and, presently, concentrate on the automation of step (b). Note that we prove single trace transformations correct. Once we have a set of heuristics that identify trace transformations, i.e., perform step (a), we may want to establish the correctness of these heuristics rather than verifying their output individually. But, for the time being, we prefer to deal with the simple semantics of traces, not with the more complicated semantics of heuristics for the transformation of traces.

More often than not, mechanical proofs are messy and full of hints to the theorem prover, e.g., hints about the use of specific lemmas, induction schemes, or variable instantiations. These hints are supplied by the user in an interactive dialogue with the theorem prover. For the proof of an isolated theorem, this is an acceptable course of action. Any (correct) tricks are welcome to make the prover succeed. However, for highly automated proofs of many similar theorems, a simple representation of the theory in the mechanized logic is of overriding concern. We must structure the path of intermediate lemmas that leads to the main theorem. We must minimize proof hints and keep those hints that are indispensible systematic. The following section develops the theory of simple sorting networks and one example proof in more detail and, at the end, reflects on our efforts to keep things clean.

4. Proving the Semantic Equivalence of Two Traces

We are applying Boyer and Moore's mechanical treatment of recursion and induction [3]. This section describes the implementation of the part of the semantic theory that is necessary to prove theorem **TAU.MAIN** in Boyer and Moore's logic, and the proof of **TAU.MAIN**. For newcomers to Boyer and Moore's logic, Section 4.1 provides an introduction adapted from [26]. Appendix A explains the types of commands (so-called 'events') that formed the proof session and contains a transcript of all functions, axioms, and lemmas established during the session. Appendix B elaborates on the central steps of the mechanical proof.

Boyer and Moore express terms functionally in a LISP-like prefix notation. To be shorter, we shall, in this document, keep basic arithmetic and logical operations in traditional infix notation.

4.1. BOYER AND MOORE'S MECHANIZED LOGIC

In Boyer and Moore's mechanized logic, as described in [3], proofs are constructed in a quantifier-free logical theory which is built on the propositional calculus with equality, variables, and function symbols. The basic theory includes four functions:

- **TRUE** and **FALSE** are both functions of zero arguments. The constants **(TRUE)** and **(FALSE)** are abbreviated **T** and **F**, respectively. The axiom **T≠F** allows them to be considered as distinct truth values.

- **EQUAL** is a function of two arguments, axiomatized to require that **(EQUAL l r)** have the value **T** or **F**, depending on whether **l=r**.
- **IF** is a function of three arguments. Axioms ensure that the value **(IF t u v)** is that of **v** if **t=F** and that of **u** otherwise.

The function **IF** allows the use of conditionals in axioms which define new functions, as in the definition

```
(AND P Q)  =  (IF P (IF Q T F) F)
```

A function capturing the semantics of each of the other logical connectives is similarly defined. Thus, corresponding to any formula ϕ, a term **p** can be constructed such that the formulas **p≠F** and ϕ have the same truth values. In this situation, the term **p** is said to be a 'theorem' if ϕ is a theorem. All formulas may thus be represented as terms.

The theory also includes

- a principle which permits the introduction of axioms specifying new types of inductively constructed objects,
- a principle for admitting axioms which define new recursive functions,
- a principle of induction (a rule of inference) based on the notion of a well-founded relation which is well-suited for inferring theorems about these objects and functions.

Two types of inductively constructed objects are relevant to the proofs presented in this paper:

- The natural numbers are formalized by the type 'number'. Peano arithmetic is realized through axioms about the functions **NUMBERP**, **ADD1**, **SUB1**, and the constant **0**. **NUMBERP** is the 'recognizer' for this type, i.e., **(NUMBERP t)** returns **T** or **F** depending on whether the value of term **t** is a number. **ADD1** is the usual successor function, and **SUB1** is its inverse function.
- Ordered pairs are formalized by the type 'list'. The functions **CONS**, **CAR**, and **CDR** are axiomatized to have the properties of the familiar LISP functions, and **LISTP** is the recognizer for this type. Finite sequences are represented by means of **CONS** and the special constant **NIL**; the sequence of terms **t1**, **t2**,..., **tn** is represented by the term

  ```
  (CONS t1 (CONS t2 (... (CONS tn NIL) ...)))
  ```

  which is abbreviated **(LIST t1 t2 ... tn)**.

The theorem proving program naturally relies heavily on induction. Several heuristics are employed to formulate induction schemes based on analysis of the structure of the recursive function definitions and the inductively constructed types which are involved in a conjecture. A proof by induction is only attempted, however, after a simplification procedure has been followed and has failed to establish the theorem.

Every conjecture which is presented to or generated by the theorem prover is first written as a single term and then represented internally as a conjunction of simpler 'clauses', each of which is a disjunction of atomic formulas called 'literals'. In order to

establish a clause, the prover first attempts to simplify each of its literals in turn, assuming the others to have the value F. A variety of heuristics is employed including the use of previously proved lemmas as rewrite rules. A term l is replaced by the term r if a rewrite rule of the form

```
(IMPLIES h (EQUAL l r))
```

is encountered and the hypothesis h can be established. The manner of use of a lemma therefore depends on its precise syntactic form.

In this paper (except Appendix A), we write (EQUAL l r) as l=r and (IMPLIES h c) as h⇒c. We also write (ADD1 n) as n+1, and (SUB1 n) as n−1 and, occasionally, (CONS t1 t2) as (t1 . t2).

4.2. TRACE REPRESENTATION

Our goal is to prove the semantic equivalence of traces tau(n) and tau⁻(n). E.g., for n=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)
```

must prove semantically equivalent to

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

We represent a trace by a list. The elements of the list are executed in sequence. If a list element is itself a list, it is called a 'parallel command' and its elements are executed in parallel. If an element of a parallel command is again a list, its elements are executed in sequence, etc. Thus, a trace is a multi-level list whose odd levels reflect sequential execution, and whose even levels reflect parallel execution. In the realm of simple sorting networks, we can represent traces as multi-level lists of integers. For example,

```
(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)
```

Note that the parallel trace TAU⁻ has a phase T1⁻ of growing concurrency,

```
(T1⁻ 5)   =  '(1 2 (3 1) (4 2))
```

followed by a phase T2⁻ of shrinking concurrency,

```
(T2⁻ 5)   =  '((5 3 1) (4 2) (3 1) 2 1)
```

In our formalism [20], parallel commands are binary, i.e., can have at most two parallel components. An n-ary parallel command is expressed as nested binary parallel commands. This coincides with Boyer and Moore's representation of a list as a nesting of pairs, as explained in the previous section. For example, the parallel command `'(5 3 1)` of trace `(TAU~ 5)` is really `'(5 . (3 . (1 . NIL)))`.

4.3. TRACE SEMANTICS

Traces have weakest precondition semantics [20]. Since a weakest precondition is a function from programs and predicates to predicates [7], the weakest precondition calculus can be directly implemented in Boyer and Moore's logic.

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. Consequently, we need not implement a complete weakest precondition generator in order to implement Stage 2. We are only interested in the equality of weakest preconditions, not in their actual values. A weakest precondition that is not affected by the trace transformations need not be spelt out but may be provided as a 'black box'. In Boyer and 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 primitive components of sorting networks are comparator modules. For the purpose of trace transformations, we are not interested in the inside of a comparator module. Therefore we declare the weakest precondition of comparator module `cs(i)` as a function

DECLARED FUNCTION: `(CS I S)`

where `S` denotes the postcondition (or 'poststate'). 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 numbers:

AXIOM `CS.TAKES.NUMBERS`: `(NOT (NUMBERP I)) ⇒ ((CS I S) = F)`

Axiom `CS.TAKES.NUMBERS` states that the prestate of `CS` for any non-number and postulate is false, i.e., that such a `CS` is not permitted. 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 prestate of any `CS` with false poststate is false, i.e., comparator modules cannot establish 'false'.

To determine the weakest precondition of some trace `L` that is composed of comparator modules `CS` for poststate `S`, we define a 'cs-machine', a function

DEFINED FUNCTION: (M.CS FLAG L S)

that composes calls to CS as prescribed by trace L. Besides L and S, 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. We can make use of the semantic declarations provided at Stage 1. The smallest component that a semantic declaration for a sorting network will mention is the comparator module. We may therefore, from Stage 1, assume knowledge about the independence of comparator modules and may express this knowledge by a declared function

DECLARED FUNCTION: (IND.CS I J)

that evaluates the independence of comparator modules cs(i) and cs(j). We then define a function

DEFINED FUNCTION: (ARE.IND.CS L1 L2)

that uses IND.CS to determine the mutual independence of all atoms of trace L1 with all atoms of trace L2. ARE.IND.CS is only interested in the atoms of a trace, not in the trace's structure. Therefore, ARE.IND.CS expects traces in a flattened form. The flattening is performed by function ALL.ATOMS, for example,

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

If the two members of a parallel command (remember the restriction to binary parallel commands) pass test ARE.IND.CS their execution has identical semantics in parallel as in sequence – only the execution time differs.

The execution time of traces plays a role in the selection of proper transformation theorems. At present, we take transformation theorems as given and only prove them by mechanical means. Therefore, execution time is left out of the current implementation.

The semantic equivalence of tau~ and tau, which we denoted by tau~ ≡ tau, is thus in Boyer and Moore's logic expressed by the term

```
(M.CS 'SEQ (TAU~ N) S) = (M.CS 'SEQ (TAU N) S)
```

4.4. TRACE COMPOSITION

For the composition of lists, we provide two functions: APPEND and APPEND2. APPEND is the regular LISP (and Boyer and Moore) append function. It handles lists that end with NIL. However, we do not require a trace to end with NIL; it may end with any atom. If trace L1 does not end with NIL, (APPEND L1 L2) drops the last atom of L1, whereas (APPEND2 L1 L2) picks it up. For example,

```
(APPEND '(1 . 2) '(3 . 4))   =  '(1 3 . 4)
(APPEND2 '(1 . 2) '(3 . 4))  =  '(1 2 3 . 4)
```

Therefore traces are composed with **APPEND2** not with **APPEND**.

The following theorem expresses the composition rule for weakest preconditions [7]:

```
LEMMA  M.CS.APPEND2.SEQ:
          (FLAG='SEQ)
            ⇒  (  (M.CS FLAG (APPEND2 L1 L2) S)
                 = (M.CS FLAG L1 (M.CS FLAG L2 S)) )
```

We subsume `M.CS.APPEND2.SEQ` by a more general theorem (`M.CS.APPEND2`).

4.5. TRACE TRANSFORMATIONS

Independence declarations are exploited via transformation rules that express commutations and parallel merges of independent program components.

The theorem for parallel merges corresponds to rule (G3i) in Section 5.2 of [20]:

```
LEMMA G3i:   (ARE.IND.CS (ALL.ATOMS L1) (ALL.ATOMS L2))
               ⇒  (  (M.CS 'PAR (CONS L1 L2) S)
                    = (M.CS 'SEQ (APPEND2 L1 (LIST L2)) S) )
```

To express commutations, we must be more specific about the meaning of 'independence'. The declaration of `IND.CS` does not provide any clues. We do not need to know everything about independence; otherwise we would define, not declare `IND.CS`. But we must be able to conclude that independent comparator modules may be commuted. As we did with `CS`, we characterize `IND.CS` by axiom:

```
AXIOM  GLOBAL.IND.CS:
          (IND.CS I J)
            ⇒  ((CS J (CS I S)) = (CS I (CS J S)))
```

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 S))
                 = (M.CS FLAG2 L2 (M.CS FLAG1 L1 S)) )
```

4.6. NON-NEIGHBORS

Stage 1 establishes that, in simple sorting networks, non-neighboring comparator modules are independent. We may provide this known fact by axiom:

```
AXIOM  NON.NEIGHBORS.ARE.IND.CS:
          (NON.NEIGHBORS I J)  ⇒  (IND.CS I J)
```

where function NON.NEIGHBORS identifies non-neighbors. NON.NEIGHBORS is defined while IND.CS is declared. With IND.CS alone we could not decide the independence of anything; with this axiom we can decide the independence of comparator modules. We may, for example, apply theorem G3i with 5 for L1 and '(3 1) for L2, since 5 is not a neighbor of 3 and 1:

```
(M.CS 'PAR '(5 3 1) S)  =  (M.CS 'SEQ '(5 (3 1)) S)
```

Two more applications of G3i, exploiting also the non-neighborhood of 3 and 1, yield:

```
(M.CS 'PAR '(5 3 1) S)  =  (M.CS 'SEQ '(5 3 1) S)
```

This formula expresses the equivalence of the parallel and sequential execution of comparator modules CS(5), CS(3), and CS(1).

4.7. SEMANTIC EQUIVALENCE OF TAU AND TAU~

The formal derivation of TAU~ from TAU requires three inductions on the length of the trace. The main induction transforms TAU and TAU~ as described in Section 2:

```
LEMMA  TAU.MAIN:   0<N  ⇒  (  (M.CS 'SEQ (TAU~ N) S)
                              = (M.CS 'SEQ (TAU  N) S) )
```

Notice the premise that trace argument N 'makes sense', i.e., is a number other than 0. (A further theorem, TAU.MAIN.COMPLETE , extends the result to meaningless N.) The prover must be made aware of the induction scheme by a hint (function RAVEL). Let us explain the induction with an example. For N=5, the following traces are semantically equivalent:

```
(TAU 5) = '(1  2 1  3 2 1  4 3 2 1  5 4 3 2 1)
 ≡ '(1 2 (3 1) (4 2)  (3 1) 2 1     5 4 3 2 1)   (ind. hyp.)
 ≡ '(1 2 (3 1) (4 2)  (5 3 1) (4 2) (3 1) 2 1)   (ind. step)
 =  (TAU~ 5)
```

The induction hypothesis is that (TAU 4) is equivalent to (TAU~ 4). The induction step 'ravels' the remaining sequential tail (SEQJ 5) to the left into (TAU~ 4). Note that the only part of TAU~ affected by this transformation is the part of shrinking concurrency, T2~. Thus we may disregard T1~ in the further proof of the induction step. To perform the transformation of the induction step, we need two more inductions. The first, named TAU.RAVEL1, uses commutations (ARE.IND.CS.IMPLIES.COMMUTATIVITY) to place comparator modules beside the parallel command that they then join (G3i) by the second induction, named TAU.RAVEL2. The intermediate stage of the trace has name UNRAVEL.T2~.

```
LEMMA  TAU.RAVEL1:
          1<N  ⇒  (  (M.CS 'SEQ (UNRAVEL.T2~ N) S)
                     = (M.CS 'SEQ (APPEND2 (T2~ N-2) (SEQJ N)) S) )

LEMMA  TAU.RAVEL2:
          0<N  ⇒  (  (M.CS 'SEQ (T2~ N) S)
                     = (M.CS 'SEQ (UNRAVEL.T2~ N) S) )
```

For example, for N=5, the following traces are semantically equivalent:

```
    (APPEND2 (T2~ 3) (SEQJ 5))
= '((3 1) 2 1  5 4 3 2 1)
≡  (UNRAVEL.T2~ 5)                    (by TAU.RAVEL1)
= '(5 (3 1)  4 (2)  3 (1)  2 1)
≡  (T2~ 5)                            (by TAU.RAVEL2)
= '((5 3 1) (4 2) (3 1) 2 1)
```

This concludes the main part of the proof. However, the prover needs some preliminary information to succeed. To enable the application of our general transformation rules G3i and ARE.IND.CS.IMPLIES.COMMUTATIVITY, we have to establish their hypotheses: the independence of certain trace parts. We also have to advise the prover of some trace identities that it cannot determine simply by opening functions. The interested reader is referred to Appendices A and B.

4.8. EVALUATION

The main objective of the mechanical proof of theorem TAU.MAIN has been to substantiate that the trace calculus of [20] has a natural representation in Boyer and Moore's logic. The functions, axioms, and theorems of this representation falls into two classes:

(a) the basic semantic theory that will be required in the proofs of many different trace transformation theorems, and
(b) the functions and lemmas pertaining to the proof of one specific transformation theorem.

To prove TAU.MAIN, we had to develop a part of the theory of sorting networks (the theory of simple sorting networks, where comparator modules deal only with adjacent elements). It includes the semantics of traces of comparator modules, the general trace transformation rules, and the theory of non-neighbors. While the basic semantic theory is constant, the theory specific to the transformation of TAU, based on the definitions of TAU and TAU~ and culminating in theorem TAU.MAIN, will recur in modified form in other trace transformations.

In the original calculus, a transformation sequence starts with a sequential trace and ends with a parallel trace. In Boyer and Moore's logic, we rewrite in the reverse direction, from parallel to sequential, in order to avoid implementation problems. We need not be aware that the prover actually transforms TAU~ and TAU, not vice versa. The direction of transformation is of no consequence to the proof of a semantic equivalence.

Because we aim at a 'methodology of proving', we are especially interested in a simple representation of the theory and in simple proof strategies. Let us explain our efforts to proceed in the most straight-forward manner.

We have reduced the number of intermediate stages of TAU's transformation to a minimum: just one intermediate stage between a step of recursive commutations and a step of recursive parallel merges. Boyer and Moore's prover cannot be expected to perform a particular succession of inductions without advice. We had to communicate three traces to the prover; the initial trace TAU, the final trace TAU~, and the intermediate trace

UNRAVEL.T2". The prover must also be made aware of the substitutions in the main induction scheme (by hint RAVEL).

Further we had to provide the prerequisites for the transformation, and some trace identities used in its proof. Say, we want to apply some theorem with certain instantiations of its variables. In Boyer and Moore's logic, we can

(a) prescribe the use of the theorem with appropriate variable instantiations as a hint for the proof in which it is needed, or
(b) provide the hypothesis of the theorem with appropriate variable instantiations as a separate lemma.

Because it is easier to establish facts independently than to tie them to specific proofs, we prefer (b) and establish as lemmas the hypotheses of the general transformation theorems being used, where free trace variables are instantiated by the particular trace parts that are subject to the transformation. We have to establish precisely these hypotheses, or the prover will not invoke the transformation theorems. For the same reason, the trace identities are expressed as *semantic* identities (with M.CS), although they hold even as *literal* identities (without M.CS). The prover would not be able to use the literal identities, because the theory deals with trace semantics, not with traces per se.

This problem is known as the 'Knuth–Bendix' problem [15]. A fact appears in Boyer and Moore's logic as a composition of functions. The same fact may be represented in many different ways depending on which function calls are opened up. Some of those representations will match hypotheses of theorems known to the prover, others will not. We have to choose a representation that is useful to the prover.

The Knuth–Bendix problem appears again in the proofs of the actual transformation theorems for TAU. The prover must be prevented from opening functions beyond the level at which the prerequisites are expressed, or it will not establish a match. In addition, since at points where a parallel merge may be applied a commutation applies as well, we have to disable the application of the first transformation (TAU.RAVEL1) in the proof of the second (TAU.RAVEL2).

The worst consequence of the Knuth–Bendix problem in the proof of TAU.MAIN is our only 'dirty' lemma: CLOSE.SEQJ. This lemma is not relevent to our theory, but it is needed to make the prover at a specific point in the proof of TAU.RAVEL1 'close' an expansion of function SEQJ (see Appendix B). To deal with problems like the necessity of lemma CLOSE.SEQJ will be one of the challenges of advancing the automation of proofs in our theory.

We required two declarations and four axioms to express the knowledge inherited from Stage 1. Of course, they are all part of the basic theory. Relative to that theory, the transformation of TAU is completely defined and certified.

5. Conclusions

Today, theorem provers are already helping to certify the mass of relatively trivial verification conditions that are generated by current mechanical program verification systems

for example [9, 14]). We study the mechanical treatment of more complicated verification conditions, conditions that are expressed recursively and proved by induction.

The goal of any research of this kind is to shift the responsibility for program development and certification from humans to expert programs. We are exploring the usefulness of techniques in automated theorem proving for the formal derivation of programs, specifically, the derivation of concurrency in programs.

At present we are focussing on a very specific class of programs: sorting networks. We chose sorting networks for three reasons. First, they are well-suited for our methodology: they terminate and only their results, not their behaviors matter. Second, they have a simple semantic structure with only one basic component: the comparator module. Third, they are extensively researched [16] and of more than academic interest: many data processing applications require sorting.

We have developed a weakest precondition generator function WP, for programs written in the language RL of our original methodology [19, 20]. The implementation of the semantic theory of RL (with function WP) would be significantly more complex than that of sorting networks presented here (with function M.CS). We want to assess the practicality of our approach in a well-understood problem domain with a simple semantic theory first.

Recently, we have extended the mechanical theory to general sorting networks with comparator modules that deal with arbitrary rather than adjacent elements. In that theory, traces are composed of pairs of integers, not single integers. We have obtained a mechanical proof of a transformation of Batcher's bitonic sort [13] and plan to continue our study of trace transformations of sorting networks. We are looking for heuristics that make our mechanical proofs depend less on interaction with the user. Essentially, we must make Boyer and Moore's prover an expert in trace transformations of sorting networks, by enriching the basic semantic theory and providing mechanical aids in the generation of intermediate lemmas.

Proving that a given trace cannot be derived is equally interesting as proving that it can be derived. For example, TAU⁻ does not represent the fastest execution that sorts in place. There are sorts with sorting networks that perform in linear time with a smaller constant and even in sublinear time [16]. However, these executions cannot be derived from TAU by the given independence declaration. The proof would have to resort to a meta theory about semantic declarations and the trace transformations they permit.

Rather than building our own structural induction prover, we decided to work with Boyer and Moore's mechanized logic. The power of Boyer and Moore's prover gives us a lot of flexibility. We can easily change the programming language that we want to work with. And we can also deal mechanically with the mathematical theory in which we qualify our semantic declarations. In the case of the insertion sort, this theory, the theory of non-neighbors, is quite simple. In other examples it may be more involved (see, for instance, the bitonic sort [13]). The structure and user-friendliness of Boyer and Moore's prover makes it a suitable tool for the development of mechanized theories.

Appendix A: Events of Proof Session

This appendix presents all events in the order in which they have been accepted by the theorem prover. We use four kinds of events: declaration of a function, definition of a function, 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 named *name* with formal arguments *args* and with body *body*. Before admission of the function, the prover attempts to certify its termination by identifying a well-founded relation such that some measure of *args* gets smaller in every recursive call. In some cases, this relation and measure must be provided in the fourth argument *hints*.

(3) Add 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.

(4) 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:

(INDUCT (*name args*)) Use the induction scheme reflected by the recursive definition of function (*name args*),

(DISABLE *ev1* . . . *evn*) Prevent the use of events *ev1* to *evn* in the proof.

The following list of events contains: 2 declared functions, 22 defined functions, 4 axioms, and 39 lemmas, i.e., 67 events in all.

SEMANTIC THEORY OF SIMPLE SORTING NETWORKS

Trace Composition

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

(DEFN APPEND2 (X Y)
      (IF (NLISTP X)
          (IF (EQUAL X NIL)
              Y
              (CONS X Y))
          (CONS (CAR X) (APPEND2 (CDR X) Y))))
```

```
(PROVE.LEMMA APPEND2.NIL (REWRITE)
      (AND (IMPLIES (NOT (EQUAL X NIL))
                    (NOT (EQUAL (APPEND2 X Y) NIL)))
           (IMPLIES (NOT (EQUAL Y NIL))
                    (NOT (EQUAL (APPEND2 X Y) NIL)))))

(DEFN PLISTP (X)
      (IF (NLISTP X)
          (EQUAL X NIL)
          (PLISTP (CDR X))))

(PROVE.LEMMA PLISTP.APPEND (REWRITE)
      (IMPLIES (AND (PLISTP X) (PLISTP Y))
               (PLISTP (APPEND X Y))))

(PROVE.LEMMA APPEND2.APPEND (REWRITE)
      (IMPLIES (PLISTP X)
               (EQUAL (APPEND2 X Y) (APPEND X Y))))

(DEFN ALL.ATOMS (L)
      (IF (LISTP L)
          (IF (EQUAL (CDR L) NIL)
              (ALL.ATOMS (CAR L))
              (APPEND (ALL.ATOMS (CAR L))
                      (ALL.ATOMS (CDR L))))
          (IF (EQUAL L NIL)
              NIL
              (LIST L))))

(PROVE.LEMMA ALL.ATOMS.PLISTP (REWRITE GENERALIZE)
      (PLISTP (ALL.ATOMS L)))

(PROVE.LEMMA ALL.ATOMS.APPEND2 (REWRITE)
      (EQUAL (ALL.ATOMS (APPEND2 X Y))
             (APPEND2 (ALL.ATOMS X) (ALL.ATOMS Y))))
```

Trace Semantics

```
(DCL CS (I S))

(DCL IND.CS (I J))

(DEFN IS.IND.CS (I L)
      (IF (NLISTP L)
          T
          (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)
          T
          (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))))

(PROVE.LEMMA ARE.IND.CS.NIL (REWRITE)
      (ARE.IND.CS L NIL))

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

(ADD.AXIOM CS.TAKES.NUMBERS (REWRITE)
    (IMPLIES (NOT (NUMBERP I))
             (EQUAL (CS I S) 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 S))))

(PROVE.LEMMA M.CS.IDENTITY (REWRITE)
      (EQUAL (M.CS FLAG (LIST (LIST L)) S)
             (M.CS FLAG L S)))

(PROVE.LEMMA M.CS.CONS (REWRITE)
      (IMPLIES (AND (NUMBERP I)
                    (OR (EQUAL FLAG 'SEQ)
                        (AND (EQUAL FLAG 'PAR)
                             (IS.IND.CS I (ALL.ATOMS L)))))
               (EQUAL (M.CS FLAG (CONS I L) S)
                      (CS I (M.CS FLAG L S)))))

(PROVE.LEMMA M.CS.APPEND2.NIL (REWRITE)
      (EQUAL (M.CS FLAG (APPEND2 L NIL) S)
             (M.CS FLAG L S))
      ((INDUCT (M.CS FLAG L S))))
```

```
(PROVE.LEMMA M.CS.APPEND2 (REWRITE)
      (IMPLIES (OR (EQUAL FLAG 'SEQ)
                   (AND (EQUAL FLAG 'PAR)
                        (ARE.IND.CS (ALL.ATOMS L1)
                                    (ALL.ATOMS L2))))
               (EQUAL (M.CS FLAG (APPEND2 L1 L2) S)
                      (M.CS FLAG L1 (M.CS FLAG L2 S))))
      ((INDUCT (M.CS FLAG L1 S))))
```

Trace Transformation Rules

```
(PROVE.LEMMA G3i (REWRITE)
      (IMPLIES (ARE.IND.CS (ALL.ATOMS L1) (ALL.ATOMS L2))
               (EQUAL (M.CS 'PAR (CONS L1 L2) S)
                      (M.CS 'SEQ (APPEND2 L1 (LIST L2)) S))))

(PROVE.LEMMA G3ii (REWRITE)
      (IMPLIES (AND (NOT (EQUAL L NIL))
                    (ARE.IND.CS (ALL.ATOMS L1) (ALL.ATOMS L2)))
               (EQUAL (M.CS 'PAR (CONS (APPEND2 L1 L) L2) S)
                      (M.CS 'SEQ (APPEND2 L1 (LIST (CONS L L2))) S)))
      ((INDUCT (APPEND2 L1 L))))

(ADD.AXIOM GLOBAL.IND.CS (REWRITE)
    (IMPLIES (IND.CS I J)
             (EQUAL (CS J (CS I S))
                    (CS I (CS J S)))))

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

(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 S))
                      (M.CS FLAG2 L2 (M.CS FLAG1 L1 S))))
      ((INDUCT (M.CS FLAG1 L1 S))))
```

Theory of Maximum

```
(DEFN MAX (L)
      (IF (NLISTP L)
          0
          (IF (LESSP (CAR L) (MAX (CDR L)))
              (MAX (CDR L))
              (CAR L))))

(DEFN NUMBER.LISTP (L)
      (IF (NLISTP L)
          (EQUAL L NIL)
          (AND (NUMBERP (CAR L))
               (NUMBER.LISTP (CDR L)))))

(PROVE.LEMMA NUMBER.LISTP.PLISTP (REWRITE)
      (IMPLIES (NUMBER.LISTP X) (PLISTP X)))
```

```
(PROVE.LEMMA MAX.NUMBER.LISTP (REWRITE)
      (IMPLIES (NUMBER.LISTP L) (NUMBERP (MAX L))))

(PROVE.LEMMA MAX.APPEND (REWRITE)
      (IMPLIES (AND (NUMBER.LISTP X) (NUMBER.LISTP Y))
               (EQUAL (MAX (APPEND X Y))
                      (MAX (LIST (MAX X) (MAX Y))))))

(PROVE.LEMMA NUMBER.LISTP.APPEND (REWRITE)
      (IMPLIES (AND (PLISTP X) (PLISTP Y))
               (EQUAL (NUMBER.LISTP (APPEND X Y))
                      (AND (NUMBER.LISTP X) (NUMBER.LISTP Y)))))
```

Theory of Non-Neighbors

```
(DEFN NON.NEIGHBORS (I J)
      (AND (NUMBERP I)
           (NUMBERP J)
           (NOT (EQUAL I J))
           (NOT (EQUAL I (ADD1 J)))
           (NOT (EQUAL (ADD1 I) J))))

(ADD.AXIOM NON.NEIGHBORS.ARE.IND.CS (REWRITE)
    (IMPLIES (NON.NEIGHBORS I J) (IND.CS I J)))

(DEFN HAS.NO.NEIGHBOR (I L)
      (IF (NLISTP L)
          T
          (AND (NON.NEIGHBORS I (CAR L))
               (HAS.NO.NEIGHBOR I (CDR L)))))

(PROVE.LEMMA HAS.NO.NEIGHBOR.IS.IND.CS (REWRITE)
      (IMPLIES (HAS.NO.NEIGHBOR I L) (IS.IND.CS I L)))

(DEFN HAVE.NO.NEIGHBOR (L1 L2)
      (IF (NLISTP L1)
          T
          (AND (HAS.NO.NEIGHBOR (CAR L1) L2)
               (HAVE.NO.NEIGHBOR (CDR L1) L2))))

(PROVE.LEMMA HAVE.NO.NEIGHBOR.ARE.IND.CS (REWRITE)
      (IMPLIES (HAVE.NO.NEIGHBOR L1 L2) (ARE.IND.CS L1 L2)))

(PROVE.LEMMA MAX.NON.NEIGHBOR (REWRITE)
      (IMPLIES (AND (NUMBER.LISTP L)
                    (LESSP (ADD1 (MAX L)) I))
               (EQUAL (HAS.NO.NEIGHBOR I L)
                      (NON.NEIGHBORS I (MAX L)))))
```

APPLICATION: EQUIVALENCE OF TAU AND TAU~

Trace Definitions

```
(DEFN SEQJ (K)
      (IF (ZEROP K)
          NIL
          (CONS K (SEQJ (SUB1 K)))))

(DEFN SEQIJ (K)
      (IF (ZEROP K)
          NIL
          (APPEND2 (SEQIJ (SUB1 K)) (SEQJ K))))

(DEFN TAU (N) (SEQIJ N))

(DEFN PARI (K)
      (IF (ZEROP K)
          NIL
          (IF (EQUAL K 1)
              (LIST 1)
              (CONS K (PARI (SUB1 (SUB1 K)))))))

(DEFN T1~ (I)
      (IF (ZEROP I)
          NIL
          (APPEND2 (T1~ (SUB1 I)) (LIST (PARI I)))))

(DEFN T2~ (I)
      (IF (ZEROP I)
          NIL
          (APPEND2 (LIST (PARI I)) (T2~ (SUB1 I)))))

(DEFN TAU~ (N)
      (APPEND2 (T1~ (SUB1 N)) (T2~ N)))

(DEFN UNRAVEL.PARI (K)
        (IF (ZEROP K)
            NIL
            (IF (EQUAL K 1)
                (LIST 1)
                (IF (EQUAL K 2)
                    (LIST 2)
                    (CONS K (LIST (PARI (SUB1 (SUB1 K)))))))))

(DEFN UNRAVEL.T2~ (I)
      (IF (ZEROP I)
          NIL
          (APPEND2 (UNRAVEL.PARI I)
                   (UNRAVEL.T2~ (SUB1 I)))))
```

Trace Transformation Prerequisites

```
(PROVE.LEMMA NUMBER.LISTP.PARI (REWRITE)
      (NUMBER.LISTP (ALL.ATOMS (PARI K))))

(PROVE.LEMMA MAX.PARI (REWRITE)
      (IMPLIES (NUMBERP K)
               (EQUAL (MAX (ALL.ATOMS (PARI K))) K)))
```

```
(PROVE.LEMMA IND.PARI (REWRITE)
      (IMPLIES (LESSP 1 K)
               (HAVE.NO.NEIGHBOR (ALL.ATOMS K)
                                 (ALL.ATOMS (PARI (SUB1 (SUB1 K)))))))

(PROVE.LEMMA NUMBER.LISTP.T2~ (REWRITE)
      (NUMBER.LISTP (ALL.ATOMS (T2~ I))))

(PROVE.LEMMA MAX.T2~ (REWRITE)
      (IMPLIES (NUMBERP I)
               (EQUAL (MAX (ALL.ATOMS (T2~ I))) I)))

(PROVE.LEMMA IND.T2~ (REWRITE)
      (IMPLIES (LESSP 1 I)
               (HAVE.NO.NEIGHBOR (ALL.ATOMS I)
                                 (ALL.ATOMS (T2~ (SUB1 (SUB1 I)))))))
```

Trace Identities

```
(PROVE.LEMMA T2~.SHIFT (REWRITE)
      (IMPLIES (LESSP 2 N)
               (EQUAL (M.CS 'SEQ
                            (UNRAVEL.PARI N)
                            (M.CS 'SEQ (T2~ (SUB1 (SUB1 (SUB1 N)))) S))
                      (CS N (M.CS 'SEQ (T2~ (SUB1 (SUB1 N))) S)))))

(PROVE.LEMMA TAU~.SHIFT (REWRITE)
      (IMPLIES (LESSP 0 N)
               (EQUAL (M.CS 'SEQ
                            (T1~ N)
                            (M.CS 'SEQ (T2~ (SUB1 N)) S))
                      (M.CS 'SEQ
                            (T1~ (SUB1 N))
                            (M.CS 'SEQ (T2~ N) S)))))
```

Trace Transformations

```
(PROVE.LEMMA CLOSE.SEQJ (REWRITE)
      (IMPLIES (NUMBERP N)
               (EQUAL (CS (ADD1 N) (M.CS 'SEQ (SEQJ N) S))
                      (M.CS 'SEQ (SEQJ (ADD1 N)) S))))

(PROVE.LEMMA TAU.RAVEL1 (REWRITE)
      (IMPLIES (LESSP 1 N)
               (EQUAL (M.CS 'SEQ (UNRAVEL.T2~ N) S)
                      (M.CS 'SEQ
                            (APPEND2 (T2~ (SUB1 (SUB1 N))) (SEQJ N))
                            S)))
      ((DISABLE SEQJ PARI T2~ UNRAVEL.PARI)))

(PROVE.LEMMA TAU.RAVEL2 (REWRITE)
      (IMPLIES (LESSP 0 N)
               (EQUAL (M.CS 'SEQ (T2~ N) S)
                      (M.CS 'SEQ (UNRAVEL.T2~ N) S)))
      ((DISABLE SEQJ TAU.RAVEL1)))
```

```
(DEFN RAVEL (N S)
      (IF (OR (ZEROP (SUB1 N))
              (NOT (NUMBERP N)))
          T
          (RAVEL (SUB1 N)
                 (M.CS 'SEQ (SEQJ N) S))))

(PROVE.LEMMA TAU.MAIN (REWRITE)
      (IMPLIES (LESSP 0 N)
               (EQUAL (M.CS 'SEQ (TAU⁻ N) S)
                      (M.CS 'SEQ (TAU N) S)))
      ((DISABLE CLOSE.SEQJ SEQJ T2⁻ UNRAVEL.T2⁻)
       (INDUCT (RAVEL N S))))
```

Extension of TAU.MAIN *to the Empty Trace*

```
(PROVE.LEMMA TAU⁻.NIL (REWRITE)
      (IMPLIES (NOT (LESSP 0 N))
               (EQUAL (TAU⁻ N) NIL)))

(PROVE.LEMMA TAU.MAIN.COMPLETE (REWRITE)
      (EQUAL (M.CS 'SEQ (TAU⁻ N) S)
             (M.CS 'SEQ (TAU N) S))
      ((DISABLE TAU⁻)))
```

Appendix B: Proof Outline of TAU's Transformation

Each transformation theorem expresses an identity of two M.CS function calls. We sketch each proof by presenting the major rewrites of the left M.CS call – these rewrites are named *L1* to *Lm*, and the right M.CS call – these rewrites are named *R1* to *Rn*. *Lm* and *Rn* yield matching calls.

TAU.RAVEL1

```
                (M.CS 'SEQ (UNRAVEL.T2⁻ N) S)
(L1)        = (M.CS 'SEQ (UNRAVEL.PARI N)
                         (M.CS 'SEQ (UNRAVEL.T2⁻ N-1) S))
(L2)        = (M.CS 'SEQ (UNRAVEL.PARI N)
                         (M.CS 'SEQ (APPEND2 (T2⁻ N-3)
                                             (SEQJ N-1)) S))
(L3)        = (M.CS 'SEQ (UNRAVEL.PARI N)
                         (M.CS 'SEQ (T2⁻ N-3)
                                    (M.CS 'SEQ (SEQJ N-1) S)))
(L4)        = (CS N (M.CS 'SEQ (T2⁻ N-2)
                              (M.CS 'SEQ (SEQJ N-1) S)))
(L5)        = (M.CS 'SEQ (T2⁻ N-2)
                         (CS N (M.CS 'SEQ (SEQJ N-1) S)))
(L6)        = (M.CS 'SEQ (T2⁻ N-2)
                         (M.CS 'SEQ (SEQJ N) S))
```

```
                (M.CS 'SEQ (APPEND2 (T2~ N-2) (SEQJ N)) S)

(R1)        = (M.CS 'SEQ (T2~ N-2)
                         (M.CS 'SEQ (SEQJ N) S))

(L1)        by UNRAVEL.T2~ and M.CS.APPEND2
(L2)        by induction hypothesis for N-1
(L3)        by M.CS.APPEND2
(L4)        by T2~.SHIFT
(L5)        by IND.T2~ and IS.IND.CS.IMPLIES.COMMUTATIVITY
(L6)        by CLOSE.SEQJ
(R1)        by M.CS.APPEND2
```

TAU.RAVEL2

```
                (M.CS 'SEQ (T2~ N) S)
(L1)        = (M.CS 'PAR (PARI N)
                         (M.CS 'SEQ (T2~ N-1) S))
(L2)        = (M.CS 'PAR (PARI N)
                         (M.CS 'SEQ (UNRAVEL.T2~ N-1) S))
(L3)        = (M.CS 'PAR (CONS N (PARI N-2))
                         (M.CS 'SEQ (UNRAVEL.T2~ N-1) S))
(L4)        = (M.CS 'SEQ (CONS N (LIST (PARI N-2)))
                         (M.CS 'SEQ (UNRAVEL.T2~ N-1) S))

                (M.CS 'SEQ (UNRAVEL.T2~ N) S)
(R1)        = (M.CS 'SEQ (UNRAVEL.PARI N)
                         (M.CS 'SEQ (UNRAVEL.T2~ N-1) S))
(R2)        = (M.CS 'SEQ (CONS N (LIST (PARI N-2)))
                         (M.CS 'SEQ (UNRAVEL.T2~ N-1) S))

(L1)        by T2~, M.CS, and M.CS.APPEND2
(L2)        by induction hypothesis for N-1
(L3)        by PARI
(L4)        by IND.PARI, G3i, and APPEND2
(R1)        by UNRAVEL.T2~ and M.CS.APPEND2
(R2)        by UNRAVEL.PARI
```

TAU.MAIN

```
                (M.CS 'SEQ (TAU~ N) S)
(L1)        = (M.CS 'SEQ (T1~ N-1)
                         (M.CS 'SEQ (T2~ N) S))
(L2)        = (M.CS 'SEQ (T1~ N-1)
                         (M.CS 'SEQ (UNRAVEL.T2~ N) S))
(L3)        = (M.CS 'SEQ (T1~ N-1)
                         (M.CS 'SEQ (T2~ N-2)
                                    (M.CS 'SEQ (SEQJ N) S)))
(L4)        = (M.CS 'SEQ (T1~ N-2)
                         (M.CS 'SEQ (T2~ N-1)
                                    (M.CS 'SEQ (SEQJ N) S)))
(L5)        = (M.CS 'SEQ (SEQIJ N-1)
                         (M.CS 'SEQ (SEQJ N) S))
```

```
                (M.CS 'SEQ (TAU N) S)
(R1)        = (M.CS 'SEQ (SEQIJ N-1)
                         (M.CS 'SEQ (SEQJ N) S))
```

(L1) by TAU⁻ and M.CS.APPEND2
(L2) by TAU.RAVEL2
(L3) by TAU.RAVEL1 and M.CS.APPEND2
(L4) by TAU⁻.SHIFT
(L5) by induction hypothesis for N-1, TAU, and TAU⁻
(R1) by TAU and SEQIJ

Acknowledgement

This research was partially supported by a Summer Research Award of the University Research Institute of the University of Texas at Austin. I am indebted to J Moore and Bob Boyer who responded patiently to my countless questions about their prover. J Moore really bore the burden of introducing me to automated theorem proving. He also helped me getting started with the implementation of my theory.

References

1. Ackerman, W. B., 'Data-flow languages', *Computer* **15**, 15–25 (1982).
2. Bates, J. L. and Constable, R. L., 'Proofs as programs', Tech. Rept. TR 82-530, Cornell University (1982).
3. Boyer, R. S. and Moore, J S., *A Computational Logic*, Academic Press, New York (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. Chang, C. and Lee, R. C., *Symbolic Logic and Mechanical Theorem Proving*, Series in Computer Science and Applied Mathematics, Academic Press, New York (1973).
6. Dahl, O. -J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming*, A.P.I.C. Studies in Data Processing, Vol. 8, Academic Press, New York (1972).
7. Dijkstra, E. W., *A Discipline of Programming*, Series in Automatic Computation, Prentice-Hall, Englewood Cliffs (1976).
8. Floyd, R. W., 'Assigning meanings to programs', *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, Vol. 19, pp. 19–32 (1967).
9. Good, D. I., 'The proof of a distributed system in Gypsy', Tech. Rept. #30, Institute for Computing Science, The University of Texas at Austin (1982).
10. Gries, D., *The Science of Programming*, Texts and Monographs in Computer Science, Springer-Verlag, New York (1981).
11. Hehner, E. C. R., 'do considered od: A contribution to the programming calculus', *Acta Informatica* **11**, 287–304 (1979).
12. Hoare, C. A. R., 'An axiomatic basis for computer programming', *Comm. ACM* **17**, 576–580, 583 (1969).
13. Huang, C. -H. and Lengauer, C., 'The automated proof of a trace transformation for a bitonic sort', Tech. Rept. TR-84-30, Department of Computer Sciences, The University of Texas at Austin (1984).
14. Johnson, S. and Nagle, J., 'Automatic program proving for real-time embedded software', *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages*, Association for Computing Machinery, pp. 48–58 (1983).
15. Knuth, D. E. and Bendix, P., 'Simple word problems in universal algebras, in *Computational Problems in Abstract Algebra*, (ed. J. Leech), Pergamon Press, London (1970).
16. Knuth, D. E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., Sect. 5.3.4 (1973).

17. Kuck, D. J., 'A survey of parallel machine organization and programming', *Computing Surveys* **9**, 29–59 (1977).
18. Lamport, L., 'The "Hoare logic" of concurrent programs', *Acta Informatica* **14**, 21–37 (1980).
19. Lengauer, C. and Hehner, E. C. R., 'A methodology for programming with concurrency: an informal presentation', *Science of Computer Programming* **2**, 1–18 (1982).
20. Lengauer, C., 'A methodology for programming with concurrency: the formalism', *Science of Computer Programming* **2**, 19–52 (1982).
21. Lengauer, C., 'A methodology for programming with concurrency', Tech. Rept. CSRG-142, Computer Systems Research Group, University of Toronto (1982).
22. 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, New York (1984). To appear.
23. Manna, Z. and Pnueli, A., 'Temporal verification of concurrent programs: the temporal framework for concurrent programs', in *The Correctness Problem in Computer Science*, (eds. R. S. Boyer and J S. Moore), International Lecture Series in Computer Science, Academic Press, New York (1981), pp. 215–273.
24. Manna, Z. and Waldinger, R., 'A deductive approach to program synthesis', *ACM TOPLAS* **2**, 90–121 (1980).
25. Owicki, S. S. and Gries, D., 'An axiomatic proof technique for parallel programs', *Acta Informatica* **6**, 319–340 (1976).
26. Russinoff, D. M., An experiment with the Boyer–Moore program verification system: a proof of Wilson's theorem', Tech. Rept. #38, Institute for Computing Science, The University of Texas at Austin (1983).