The Static Derivation of Concurrency and Its Mechanized Certification

Christian Lengauer
Chua-Huang Huang
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

This is an attempt to combine the two research areas of programming methodology and automated theorem proving. We investigate the potential for automation of a programming methodology that supports the compile-time derivation of concurrency in imperative programs. In this methodology, concurrency is identified by the declaration of certain semantic properties (so-called "semantic relations") of appropriate program parts. Semantic declarations can be exploited to transform the sequential execution of the program into a parallel execution. We make observations about the automation of correctness proofs of such transformations for a limited domain of programs: sorting networks.

1. Introduction

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

Automating or, more exactly, mechanically certifying the derivation of programs helps both the programmer and the customer who uses the program. The programmer will find an automated derivation more difficult and tedious than a derivation by hand. This is to be expected: an automated derivation does not permit any informal steps; each ever so little detail has to be formalized. However, what is gained, is the near-to-complete confidence that the

derivation rules have been applied correctly. The customer reaps most of the reward of an automated derivation. All he has to believe in order to be convinced of the correctness of the programmer's product is:

(a) that the program's specification meets his needs, and

(b) that the theorem which states that the program satisfies the specification is correctly represented in the mechanized programming calculus.

He does not have to be concerned with any aspects of the proof at all. However, both the programmer and the customer must believe one more thing: that the programming calculus has been implemented correctly, i.e., that no faulty programs can be certified.

The methodology in whose automation we are interested focusses on the static derivation of concurrency in imperative programs [9]. In this methodology, the derivation of concurrency proceeds by a successive compression of the program's executions based on the declaration of certain useful program properties. Most interesting programs contain recursions or loops. The most effective and practical transformations of such programs will also be recursive, and their proofs of correctness will require induction. We are therefore interested in the mechanical treatment of recursion and induction.

The following section reviews our methodology. Sect. 3 introduces the class of programs that we explore: sorting networks. After some general observations about the mechanical support of trace transformations and a justification why we view them as theorems (Sect. 4), we describe a mechanically supported "proof methodology" of trace transformations and illustrate it on several sorting networks (Sect. 5). We conclude the paper with a discussion of the challenges in the automation of this proof methodology (Sect. 6).

A more detailed account of our mechanized semantic theory and the full description of a mechanical proof can be found in [12].

2. A Methodology for the Static Derivation of Concurrency

Our goal is to mechanize parts of a particular methodology for the derivation of concurrency in programs [9]. 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 dis-

tributed 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.

This methodology can be applied to every programming problem that is completely specified by an input/output assertion pair. A terminating solution must exist, i.e., the output assertion must not be false. An execution time limit in form of a function of the input variables may or may not be added. One could conceive also the addition a storage space limit but, in its present from, the methodology does not provide for that.

The methodology cannot be applied to a programming problem with additional constraints like a specific concurrent behavior. Programs with a specific behavior can be derived (see, for instance, the Producer/Consumer and the Dining Philosophers in [9]), but the correctness of such behavior has to be argued informally.

Thus, we permit the specification of a programming problem in three parts:

(a) the input constraints under which the program shall operate,

(b) the results which the program is supposed to achieve, and

(c) an optional 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 a total order of its operations, 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 of program S is, for instance, S: S1;S2. The semicolon denotes "application". It may be implemented by executing S1 and then S2, but need not be in all

cases.

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

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

To declare semantic relations for some program, one does not need to understand the program as a whole. A local understanding of the components appearing in the declared relation is sufficient. The concurrency that is induced by semantic declarations is of a very simple nature: there is no need for synchronization (other than at the point of termination) or mutual exclusion, as is required for conventional concurrent processes. 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.

For concurrency to be correct, a program has to fulfill intricate requirements. That is what makes concurrency so hard to understand. It is easier to derive concurrency on an informed basis (as the last step of the program derivation) than on an uninformed basis (as the first step of the program derivation). The correctness proof of concurrency is easier at a refinement level where concurrency is simple, e.g., between two independent program parts than at a refinement level where concurrency is complicated, e.g., between two processes that require synchronization and mutual exclusion.

Thus, in our methodology, 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 by the usual program combinators, e.g., composition: S1;S2 (read: "S2 is applied to the results of S1").

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

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, 13]. Automation of Stage 2 would yield a very powerful optimizing compiler (since we view concurrency as optimization). Early work in this area [8] 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 formalism, 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. Expository Domain: Sorting Networks

Semantic relations can be declared for programs in any imperative programming language that has a weakest precondition semantics. For the purpose of our investigation we choose a very simple language. We do not want to complicate our mechanical proofs of trace

transformations by unduly complicated semantics of programs and traces. We define the language of *sorting networks* [7]. The general problem that we pursue is to sort an array $a_{0..n}$ of numbers into ascending order in no more time than O(n). The linear time requirement forces us to consider a concurrent execution. In the language of sorting networks, refinements can have the following structure:

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

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

(3) The *composition* S1;S2 of refinements S1 and S2 applies S2 to the results of S1.

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

Since we are concerned with the trace calculus of the methodology, we do not dwell on the refinement of programs but accept the particular sorting network whose trace transformations we want to study as given. So far, we have studied three sorting networks: the insertion sort, the odd-even transposition sort, and the bitonic sort [7]. The insertion sort and the odd-even transposition sort can be expressed as simple sorting networks. The bitonic sort expects array a already presorted in bitonic order. Let us describe each of the three sorting networks in turn.

3.1. Insertion Sort

The following refinement describes the insertion sort:

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

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

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

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

do not share variables may be declared independent.

```
                    !cs(i)
|i-j|>1   ⇒   cs(i)||cs(j)
```

Note that the prerequisite `|i-j|>1` makes `cs(i)||cs(j)` a semantic rather than syntactic condition. (Semantic declarations can also be qualified with respect to a postcondition. For the underlying theory see [11].)

For, say, a six-element array (n=5), the refinement has the following sequential execution, if we interpret composition ';' as execution in order '→' and expand components `sort(i)` and `S(i)` (i≤5) of `sort(5)`:

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

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

```
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)
```

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

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

3.2. Odd-Even Transposition Sort

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

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

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

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

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

$$!\mathtt{cs(i)}$$

$$|\mathtt{i-j}|>1 \;\Rightarrow\; \mathtt{cs(i)}\,||\,\mathtt{cs(j)}$$

The sequential trace of this refinement for a five-element array (n=4) is:

$$\mathtt{tau(4)} = \mathtt{cs(3)}\to\mathtt{cs(1)}\to\mathtt{cs(4)}\to\mathtt{cs(2)}\to\mathtt{cs(3)}\to\mathtt{cs(1)}\to\mathtt{cs(4)}\to\mathtt{cs(2)}\to\mathtt{cs(3)}\to\mathtt{cs(1)}$$

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

$$\mathtt{tau^{\sim}(4)} = \left\langle \begin{matrix}\mathtt{cs(1)}\\ \mathtt{cs(3)}\end{matrix} \right\rangle \to \left\langle \begin{matrix}\mathtt{cs(2)}\\ \mathtt{cs(4)}\end{matrix} \right\rangle \to \left\langle \begin{matrix}\mathtt{cs(1)}\\ \mathtt{cs(3)}\end{matrix} \right\rangle \to \left\langle \begin{matrix}\mathtt{cs(2)}\\ \mathtt{cs(4)}\end{matrix} \right\rangle \to \left\langle \begin{matrix}\mathtt{cs(1)}\\ \mathtt{cs(3)}\end{matrix} \right\rangle$$

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

3.3. Bitonic Sort

An array $a_{0..n}$ is in *bitonic order* if $a_0 \geq \ldots \geq a_i \leq \ldots \leq a_n$ for some $0 \leq i \leq n$. Let us write array $a_{0..n}$ as a sequence $\langle a_0, a_1, \ldots, a_n \rangle$. The bitonic sorting algorithm sorts an array `a` that is already in bitonic order into ascending order by sorting the subsequences $\langle a_0, a_2, \ldots \rangle$ and $\langle a_1, a_3, \ldots \rangle$ independently, and then comparing and interchanging (a_0, a_1), (a_2, a_3),... . Since the subsequences of a bitonic sequence are also bitonic, $\langle a_0, a_2, \ldots \rangle$ and $\langle a_1, a_3, \ldots \rangle$ can be sorted by the same algorithm, until all subsequences have length 1. The bitonic sort is not a simple sorting network. It requires the general comparator module `cs(i,j)`.

The refinement of the bitonic sort is:

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

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

          S(base, dist, step, 0):       skip
(leng>0)  S(base, dist, step, leng):    cs(base, base+dist);
                                        S(base+step, dist, step, leng-1)
```

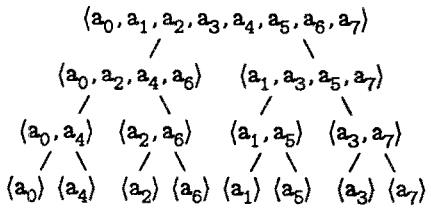
Refinement **sort** performs the bitonic sort as described. It is qualified by three parameters, **base**, **step**, and **leng**, that identify a subsequence of **a**: **base** is the index of the first element, **step** is the difference of the indices of any two adjacent elements, and **leng** is the number of elements in the subsequence. Refinement **S** performs the step of comparisons and interchanges. It is qualified by four parameters, **base**, **dist**, **step**, and **leng**, that identify a sequence of comparator modules that access array **a**: **base** is the index of the left array element accessed by the first comparator module, **dist** is the distance of the left and right elements accessed by any comparator module, **step** is the distance of the left elements (or right elements) of any two adjacent comparator modules, and **leng** is the number of comparator modules in sequence.

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

$$!\mathtt{cs}(i_1,i_2)$$

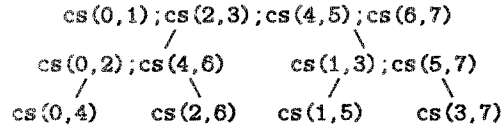
$$i_1 \neq j_1 \wedge i_1 \neq j_2 \wedge i_2 \neq j_1 \wedge i_2 \neq j_2 \;\Rightarrow\; \mathtt{cs}(i_1,i_2) \,||\, \mathtt{cs}(j_1,j_2)$$

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



At each node $\langle a_{i_1}, a_{i_2}, a_{i_3}, a_{i_4}, \ldots \rangle$, the bitonic sorting algorithm requires an application of

comparator modules $cs(i_1,i_2)$, $cs(i_3,i_4)$, ..., which we shall call a *segment*. The following *segment tree* corresponds to the previous sequence tree:



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

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

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

tau(7) has length 12. In general, $tau(2^k-1)$ has length $2^{k-1}k$. (The refinement works for all bitonic arrays, but we choose to consider only arrays whose length is a power k of 2. Such arrays yield complete sequence and segment trees.) Observe that any two distinct segments **x** and **y** in the segment tree which are not in an ascendant/descendant relationship have no common elements. Such **x** and **y** are independent, and we can commute them or make them parallel. For instance, we can commute all segments that are on the same level in the tree (i.e., that have the same distance from the root) into adjacency:

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

Then we can merge each level into one parallel command:

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

tau~(7) is of length 3, with a concurrency degree of 4. In general, $tau^{\sim}(2^k-1)$ is of length k, with a concurrency degree of 2^{k-1}.

4. On the Mechanical Support of Trace Transformations

Given a sequential trace that we know to be correct, we would like to derive an equivalent but faster parallel trace. Let us assume a recursive sequential trace. We can prove its equivalence with the parallel trace by a recursive application of a sequence of trace transformations. Although such trace transformations are in many cases quite simply described in in-

formal English, their formal application is extremely tedious (as is effectively demonstrated by our manually derived proof of the insertion sort transformation in Sect. 5.4 of [11]). We do not want to rely on an informal description but would like some mechanical aid in the formal application.

We might be tempted to view the trace transformation as a recursive algorithm. Say, algorithm `trans(n)` transforms sequential trace `tau(n)` into parallel trace `tau~(n)` by appropriately commuting and ravelling `tau`'s comparator modules. The computational complexity of `trans(n)` will depend on the particular transformation it performs. For instance, [12] contains a cubic algorithm for the transformation of the insertion sort. If we intend to sort frequently it is very reasonable to "buy" a linear execution with cubic compilation. However, the algorithmic approach to transformation has one fundamental problem: an unbounded trace can never be completely transformed in finite time - and recursive or looping programs yield unbounded traces.

A better approach is to treat trace transformations as theorems, not algorithms. A trace transformation theorem states the semantic equivalence of a sequential trace and its parallel transformation:

```
semantics of parallel trace = semantics of sequential trace
```

In particular, recursive transformations are inductive theorems. Transformation theorems of sorting networks are of the form:

```
TAU.MAIN:   For all n>0,
            semantics of tau~(n) = semantics of tau(n)
```

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. In other words, the length of the proof does not depend on the length of the trace.

Our current focus is the automation of such proofs. For this purpose, we use a powerful induction prover [4] that is based on a mechanized functional logic particularly suitable for program verification [3]. The prover is designed to prove theorems about recursive functions but is not an expert on sorting networks and their trace transformations. Our attempts to turn it into such an expert are described in the following section.

Ultimately we would like the mechanical support not only in the *proof* but also in the *discovery* of transformation theorems. We imagine a set of mechanized heuristics that transform sequential traces correctly into equivalent parallel traces, using induction. A formal correctness proof of these heuristics would save us from proving the transformation of every single trace separately. However, for the time being, we prefer to deal with the simple seman-

tics of traces, not with the more complicated semantics of heuristics for the transformation of traces.

5. The Mechanical Correctness Proof of Trace Transformations

We are applying Boyer & Moore's mechanical treatment of recursion and induction [3]. All the reader has to know about Boyer & Moore's mechanized logic to understand this paper is that terms in first-order predicate logic are expressed in a LISP-like functional form. (We will here actually keep basic logic and arithmetic operations in infix notation.) Predicates are functions with a boolean range. There are no quantifiers. A variable that appears free in a term is taken as universally quantified. For example, the term

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

expresses the fact that any number is smaller than the same number incremented by 1. Functions can be declared (without a function body) or defined (with a function body), and facts can be asserted (introduced as an "axiom") or proved (introduced as a "lemma").

This section sketches the implementation of the semantic theory that is necessary to prove trace transformation theorems for sorting networks in Boyer & Moore's logic. We shall gloss over a lot of details. For instance, we shall not display the bodies of the defined functions that we introduce.

5.1. Trace Representation

We represent a trace by a LISP 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 by multi-level lists of integers. For example, traces tau(5) and tau~(5) of the insertion sort,

$$\begin{aligned}\text{tau}(5) \;=\; & \text{cs}(1)\rightarrow\text{cs}(2)\rightarrow\text{cs}(1)\\ & \rightarrow\text{cs}(3)\rightarrow\text{cs}(2)\rightarrow\text{cs}(1)\\ & \rightarrow\text{cs}(4)\rightarrow\text{cs}(3)\rightarrow\text{cs}(2)\rightarrow\text{cs}(1)\\ & \rightarrow\text{cs}(5)\rightarrow\text{cs}(4)\rightarrow\text{cs}(3)\rightarrow\text{cs}(2)\rightarrow\text{cs}(1)\end{aligned}$$

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

are represented by

```
(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)
```

In our formalism [10], 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 LISP's (and Boyer & Moore's) representation of a list as a nesting of pairs. E.g., the parallel command '(5 3 1) of trace (TAU~ 5) is really '(5 . (3 . (1 . NIL))).

In the realm of general sorting networks, traces are represented as multi-level lists of pairs of integers.

5.2. Trace Semantics

Traces have weakest precondition semantics [10]. Since a weakest precondition is a function from programs and predicates to predicates [5], the weakest precondition calculus can be directly implemented in Boyer & 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 & 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 a comparator module cs as a function

Declared Function: (CS I S)

where I represents an integer if cs is simple and a pair of integers if cs is general, and 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 simple 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 poststate is false, i.e., that such a CS is not permitted. A respective axiom for general comparator modules tests for pairs of numbers rather than numbers. The other axiom expresses the "rule of the excluded miracle" (Dijkstra's first healthiness criterion [5]) 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. Beside 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 I and J. Again, look at I and J as integers or pairs of integers, as appropriate. We then define a function

Defined Function: (ARE.IND.CS L1 L2)

that uses IND.CS to determine the mutual independence of all comparator modules of trace L1 with all comparator modules of trace L2. 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 their 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 for any of the three previously described transformations is formally expressed as

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

5.3. 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 transformation rule (G3i) of Sect. 5.2 of [10]:

```
Lemma G31:   (ARE.IND.CS L1 L2)
               ⇒  (  (M.CS 'SEQ <L1 L2> S)
                    = (M.CS 'SEQ  L1→L2  S))
```

For clarity, we return here to our previous notation for traces. Traces must, of course, be fully represented in the mechanized logic.

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

5.4. Independence Criteria

For simple sorting networks, we have introduced the concept of "non-neighbors" to declare independence. Two simple comparator modules are *non-neighbors* if their indices differ by at least 2. 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 simple comparator modules. We may, for example, apply theorem G3i with cs(5) for L1 and cs(3)→cs(1) for L2, since cs(5) is not neighbor of cs(3) and cs(1):

```
(M.CS 'SEQ <cs(5) cs(3) cs(1)> S)  =  (M.CS 'SEQ <cs(5) cs(3)→cs(1)> S)
```

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

```
(M.CS 'SEQ <cs(5) cs(3) cs(1)> S)  =  (M.CS 'SEQ cs(5)→cs(3)→cs(1) S)
```

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

For general sorting networks, we characterize independence by the concept of "non-overlap". Two general comparator modules *do not overlap*, if they do not touch the same array element. This fact is provided by axiom:

```
Axiom NO.OVERLAP.ARE.IND.CS:
                (NO.OVERLAP I J)  ⇒  (IND.CS I J)
```

where function NO.OVERLAP establishes non-overlap.

5.5. Application Theorems

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

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

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

Up to this point, we have described the implementation of the basic semantic theory, the

part that applies to all simple, resp., general sorting networks. It consists of the semantics of traces of comparator modules, a set of trace transformation theorems, and an independence criterion for comparator modules. The semantic theory is not fully represented in the mechanized logic: we introduced two declared (not defined) functions. The theory is also not fully certified: we made four axiomatic assumptions. They reflect the knowledge that is presumed in the theory.

Even with the basic semantic theory in place and after proper definition of the initial trace TAU and the final trace TAU⁻, the work required to make the proof of an application TAU.MAIN succeed is substantial. Essentially, we have to communicate our proof strategy to the prover. Where the transformation consists of several steps, the prover may have to be informed about each individual step. For instance, since we can commute at any place where we can merge (remember that independence implies commutativity), we must tell the prover about the transformation that we prefer: commutation or merge. Our transformations of the insertion sort and the bitonic sort each consist of two steps: one of commutations and one of merges. The transformation of the odd-even sort consists of only one step of merges. For every step of the transformation, the trace parts that are manipulated must be identified, and their independence must be established. This generally involves educating the prover about useful facts of number theory. For our simple sorting networks, we had to tell the prover about properties of maximization, for our general sorting network about properties of division. Establishing these prerequisites before the proof of the application theorem is the most tedious aspect of a mechanized certification. For an effective use of a mechanized theory in many applications, clean and widely applicable proof strategies are of central importance.

		all comparator modules		
BASIC THEORY		trace semantics trace transformation rules		
		simple comp. mods.		*general comp. mods.*
	independence criterion	non-neighbors		no overlap
		insertion sort	*odd-even sort*	*bitonic sort*
APPLICATION	*algebraic prerequisites*	maximization	maximization	division
	transformation strategy	1st step: commute 2nd step: merge	one step: merge	1st step: commute 2nd step: merge
	auxiliary lemmas	see [12]		see [6]
	main theorem	TAU.MAIN	TAU.MAIN	TAU.MAIN

We shall provide no further details of the individual proofs of our three applications. The previous table displays the overall proof structure. The proof of the insertion sort is

documented in [12], that of the bitonic sort in [6].

While the basic theory may contain some declared functions and axioms (and our's does), the application part of the proof should not (and ours do not). That is, with respect to the basic theory, applications should be completely certified. It is important that every axiomatic assumption is fully understood. An inconsistency in an axiom is not recognized by the prover and puts the entire mechanized theory into jeopardy!

6. Conclusions

By its very name, the area of automated theorem proving invites high expectations: the hope is kindled that, whenever the human prover seems lost or uncertain in a proof, the mechanism will take over and guide him along. A presently more fitting name would be *automated proof checking*: the human has to conceive and carry out the proof; but he can count on a mechanized certification of his proof steps, if these steps are chosen appropriately. In order to make the mechanized certification succeed, the human prover has to be familiar not only with the abstract theory on which his proof relies but also with its mechanized counterpart. Like it is the crux of numerical analysis that floating point numbers do not have the nice properties of real numbers, it is the dilemma of automated theorem proving that the mechanization of a logic does not preserve many of its desirable properties. Therefore, a proof certified by a mechanism is actually more difficult than a proof certified by a human. But it is also more reliable.

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

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

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

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

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

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

(d) aa+bb = cc

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

One major concern of automated certification is execution efficiency. The most fundamental efficiency requirement is termination. An inappropriate choice of proof steps may lead to an infinite computation. For instance, many automated provers, like Boyer & Moore's, rewrite equalities only in one direction in order to avoid infinite looping. E.g., with the knowledge of A=B, Boyer & Moore's prover will substitute B for A in proofs, but not vice versa. This has immediate consequences for the implementation of our theory: semantic declarations may be exploited only in one direction. In any particular proof, we may commute left or commute right, but not both; we may use idempotence to compress traces or expand traces, but not both; we may increase or decrease the parallelism in a trace, but not both. Even if we stick with one direction, we may have termination problems if our transformation sequence is not well-founded. For instance, decreasing parallelism is always well-founded, while increasing parallelism is not. Therefore, we actually let the prover transform traces "backwards", from parallel to sequential.

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

Acknowledgements

We are grateful to J Moore and Bob Boyer who patiently answered our countless questions about their prover.

References

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

2. Bates, J. L., and Constable, R. L. Proofs as Programs. Tech. Rept. TR 82-530, Cornell University, 1982.

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

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

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

6. 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.

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

8. Kuck, D. J. "A Survey of Parallel Machine Organization and Programming." *Computing Surveys 9*, 1 (Mar. 1977), 29-59.

9. 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:

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

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

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

13. Manna, Z., and Waldinger, R. "A Deductive Approach to Program Synthesis." *ACM TOPLAS 2*, 1 (Jan. 1980), 90-121.