A Mechanically Checked Proof of a Comparator Sort Algorithm

Bishop Brock*
J Strother Moore†

February 22, 1999

Abstract

We describe a mechanically checked correctness proof for the comparator sort algorithm underlying a microcode program in a commercially designed digital signal processing chip. The abstract algorithm uses an unlimited number of systolic comparator modules to sort a stream of data. In addition to proving that the algorithm produces an ordered permutation of its input, we prove two theorems that are important to verifying the microcode implementation. These theorems describe how positive and negative "infinities" can be streamed into the array of comparators to achieve certain effects. Interesting generalizations are necessary in order to prove these theorems inductively. The mechanical proofs were carried out with the ACL2 theorem prover. We find these proofs both mathematically interesting and illustrative of the kind of mathematics that must be done to verify software.

1 Informal Discussion of the Problem

It is often necessary to perform statistical filtering and peak location in digital spectra for communications signal processing. In this paper we consider an abstraction of the algorithm implemented on one such microprocessor, the Motorola CAP digital signal processor [5]. One of the major functional units of the CAP is the adder array, a collection of 20-bit adder/subtracters, each of which has 8 dedicated input registers and a dedicated path to a local memory. The CAP adder array was originally designed to support fast FFT computations, but the designers also included the datapaths necessary to accelerate peak finding.

*IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78756, brock@austin.ibm.com

†Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

The so-called "5PEAK" program of the CAP [3] uses the microprocessor's adder array as a systolic comparator array as shown in Figure 1. The program streams data through the comparator array and finds the five largest data points and the five corresponding memory addresses. In this informal discussion we

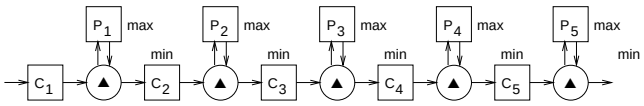

Figure 1: Abstract View of Comparator Array

largely ignore memory addresses paired with each data point.

Candidate data points enter the array by way of register C_1 and move through the array, towards the right in the diagram. Maximum values (peaks) remain in the array in the P_n registers, and the minima are eventually discarded when they pass out of the last comparator. On each cycle the comparator array updates the registers as follows:

$$\begin{aligned} C_1 &= \text{next data point}, \\ C_n &= \min(C_{n-1}, P_{n-1}), \quad n > 1, \\ P_n &= \max(C_n, P_n). \end{aligned}$$

Informally, the peak registers, P_i, maintain the maximum value that has passed by that point in the comparator array. How can we use this array to sort? Or, more particularly, to identify the n highest peaks in the stream of data?

Using the comparator array to find the five maxima requires several steps. We explain the algorithm by example here. In the following we will represent the contents of the comparator array registers in the format shown below, with the contents of each peak register above the contents of the corresponsing candidate register.

$$\begin{bmatrix} P_1 & P_2 & P_3 & P_4 & P_5 \\ C_1 & C_2 & C_3 & C_4 & C_5 \end{bmatrix}$$

The next state of the comparator array on each cycle is

$$\begin{bmatrix} \max(P_1, C_1) & \max(P_2, C_2) & \max(P_3, C_3) & \max(P_4, C_4) & \max(P_5, C_5) \\ d & \min(P_1, C_1) & \min(P_2, C_2) & \min(P_3, C_3) & \min(P_4, C_4) \end{bmatrix},$$

where d represents the next data point.

We will illustrate the peak search for the 10-element data vector

$$\begin{bmatrix} 2 & 9 & 3 & 5 & 4 & 1 & 8 & 7 & 10 & 6 \end{bmatrix}.$$

Although this example uses small unsigned numbers for simplicity, the CAP implementation of comparator array and `5PEAK` microcode will correctly search

any vector of signed, 20-bit 2's complement data values, subject to a few obvious restrictions.

The comparator array is initialized by setting C_1 to the first (leftmost) element of the data vector, and setting every other register to $-\infty$. In the fixed bit-width hardware realization of the comparator array on the CAP, the role of $-\infty$ is played by the most negative number, -2^{19}.

$$\begin{bmatrix} -\infty & -\infty & -\infty & -\infty & -\infty \\ 2 & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

Carrying out the comparator array operations for 9 more steps results in the configuration

$$\begin{bmatrix} 10 & 8 & 5 & 3 & 2 \\ 6 & 9 & 7 & 4 & 1 \end{bmatrix}.$$

Up to this point the algorithm described here is essentially identical to the two VLSI sorting algorithms described in [8, 4]. These researchers offered special-purpose hardware designs with the same basic compare-exchange step described above. There is a key difference, however, in that the VLSI sorting proposals require reversing the direction of data flow to extract the sorted data. In these approaches the sorting machine is a stack that accepts data pushes in arbitrary order but pops data in sorted order. For example, if after loading the sample vector we were to redefine the next-state function of our sorting array to be

$$\begin{aligned} Output &= max(C_1, P_1), \\ P_n &= max(C_{n+1}, P_{n+1}), \\ C_n &= min(C_n, P_n), \quad n < 5, \\ P_5 &= -\infty \end{aligned}$$

and pump ten times, the original input vector would be popped to the output in descending order. As long as there are enough registers and comparators for the input data set size, a machine of this type can sort data as fast as it can be physically moved to and from the sorting array. Reference [8] also describes ways to pipeline the use of these sorting machines to increase throughput.

Although the CAP provided numerous data paths in the adder array, reversing the direction of data flow was not possible, and another solution to extracting the maxima had to be found. Among the many possiblities that were supported by the hardware, the most straightforward involved simply continuing to step the original compare-exchange algorithm and collecting the maxima as they are ejected out the array. This was the algorithm ultimately encoded in CAP microcode. In the CAP algorithm, data input is completed by stepping the array one more cycle with a dummy input of $+\infty$. In the fixed bit-width hardware realization on the CAP the role of $+\infty$ is played by the most positive data value,

$2^{19} - 1$.

$$\begin{bmatrix} 10 & 9 & 7 & 4 & 2 \\ +\infty & 6 & 8 & 5 & 3 \end{bmatrix}.$$

At this point register P_1 holds the maximum value, yet the rest of the array is not yet ordered in any discernable way, except that the P_n registers satisfy the invariant given above. As we will show later, this invariant guarantees that if we 'pump' the array four times with $+\infty$, then the maxima will collect at the end of the array in registers P_3, C_4, P_4, C_5, and P_5.

$$\begin{bmatrix} +\infty & +\infty & 10 & 8 & 6 \\ +\infty & +\infty & +\infty & 9 & 7 \end{bmatrix}.$$

At this point the comparator array data registers $C_1, P_1, C_2, \ldots, C_5, P_5$ are ordered, and the array acts like a shift register as long as $+\infty$ is pumped into C_1. Pumping the array five times with $+\infty$ forces the five maxima out of the comparator array in reverse order, where they can be collected and stored.

To summarize, the systolic comparator array can be used to compute the five maxima of a data vector by the following steps:

- The first data point is loaded into C_1, and the rest of the comparator array is initialized to $-\infty$.
- The data vector is pumped into the array one point at a time, and a single $+\infty$ is inserted to finish the data input.
- Pumping four times with $+\infty$ causes the maxima to collect at the end of the array.
- Pumping five times with $+\infty$ forces the maxima out of the array in reverse order, where they are collected and stored.

The algorithm above is implemented in microcode on the CAP. It is among several microcode programs for that processor that we have mechanically verified. As described briefly in [3], we formalized the CAP in the ACL2 logic, sketched below. We then extracted the microcode for the `5PEAK` program from the CAP ROM, obtaining a sequence of bit vectors, and used the ACL2 theorem prover to show that when the abstract CAP machine executes the extracted code on an appropriate initial state and for the appropriate number of cycles, the five highest peaks and their addresses are deposited into certain locations. We defined the "highest peaks and their addresses" by defining, for specification purposes only, a sort function in ACL2 which sorts such address/data pairs into descending order. The reader will see that this sorting function is exactly the stack-like sorting method of the VLSI implementations described above. In our `5PEAK` specification we refer to the first five pairs in the ordering.

The argument that the microcode is correct is quite subtle, in part because an arbitrary amount of data is streamed through and in part because the positive

and negative infinities involved in the algorithm can be legitimate data values but are accompanied by bogus addresses; correctness depends on a certain "anti-stability" property of the comparator array. A wonderfully subtle generalization of a key lemma was necessary in order to produce a theorem that could be proved by mathematical induction.

In this paper we discuss only the high-level algorithm sketched above and its correctness proof. We do not discuss the microcode itself.

The event list is available at http://www.cs.utexas.edu/users/moore/publications/csort/csort.lisp.

2 ACL2

Before we present this work in detail we briefly describe the ACL2 logic and theorem prover.

ACL2 stands for "A Computational Logic for Applicative Common Lisp." ACL2 is both a mathematical logic and system of mechanical tools which can be used to construct proofs in the logic. The logic formalizes a subset of Common Lisp. The ACL2 system is essentially a re-implemented extension, for applicative Common Lisp, of the so-called "Boyer-Moore theorem prover" Nqthm [1, 2].

The ACL2 logic is a first-order, essentially quantifier-free logic of total recursive functions providing mathematical induction and two extension principles: one for recursive definition and one for "encapsulation."

The syntax of ACL2 is a subset of that of Common Lisp. However, we do not use Lisp syntax in this paper. The rules of inference are those of propositional calculus with equality together with instantiation and mathematical induction on the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$. The axioms of ACL2 describe five primitive data types: the numbers (actually, the complex rationals), characters, strings, symbols, and ordered pairs or "conses".

Essentially all of the Common Lisp functions on the above data types are axiomatized or defined as functions or macros in ACL2. By "Common Lisp functions" here we mean the programs specified in [9] that are (i) applicative, (ii) not dependent on state, implicit parameters, or data types other than those in ACL2, and (iii) completely specified, unambiguously, in a host-independent manner. Approximately 170 such functions are axiomatized or defined. The functions used in Table 1 are particularly important here.

Common Lisp functions are partial; they are not defined for all possible inputs. In ACL2 we complete the domains of the Common Lisp functions and provide a "guard mechanism" by which one can establish that the completion process does not affect the value of a given expression. See [6].

The most important data structure we use in this paper is lists. The empty list is usually represented by the symbol **nil**. The non-empty list whose first element is x and whose remaining elements are those in the list y is represented

expression	*meaning*
endp (x)	true iff x is the empty list
cons (x, y)	the ordered pair $< x, y >$
car (x)	the left component of (the ordered pair) x
cdr (x)	the right component of x
cadr (x)	the left component of the right component of x
cddr (x)	the right component of the right component of x
zp (x)	$x = 0$ (or x is not a natural number)
len (x)	the number of elements in the list x

Table 1: The Meaning of Certain Expressions

by the ordered pair $< x, y >$. This ordered pair is the value of the expression cons (x, y).

Here is an example of a simple list processing function, namely, the function for concatenating two lists. In the syntax of Common Lisp we could write this as

```
(defun append (x y)
  (if (endp x)
      y
      (cons (car x) (append (cdr x) y)))).
```

but we will here use the notation

DEFINITION:
append (x, y)
=
if endp (x) **then** y
else cons (car (x),
append (cdr (x), y))
fi

The concatenation of the empty x to y yeilds y. The concatenation of a non-empty x to y is obtained by consing the first element of x, car (x), to the concatenation of the rest of x, cdr (x), to y.

Readers interested in learning more about Common Lisp should consult [9]. Readers interested in the logical foundations of applicative Common Lisp as formalized in ACL2 should see [7]. Readers interested in the ACL2 system should see [6, 3] as well as the home page for ACL2, http://www.cs.utexas.edu/users/-moore/acl2, which contains the source code, 5 megabytes of hypertext documentation, a bibliography, and many applications.

3 High-Level Specification

We typically approach the verification of a machine code program in two phases, by first characterizing what the machine code computes at a low level, and then showing that the low-level behavior meets, or is somehow equivalent to, a higher-level specification. If we had approached the veriication of the `5PEAK` algorithm in this way we would have first specified the exact function computed by the comparator array, and then proved that this function computed the five peaks. For this particular application, it turned out to be more convenient to directly prove that the machine code execution satisfies the high-level specification, but to formalize the specification in a way particularly oriented toward the code. To that end, we specified the `5PEAK` application in terms of an abstract sorting algorithm. We proved that the `5PEAK` microcode computes the first five elements of the output vector of this abstract algorithm, i.e., the five maxima.

We defined the abstract sorting algorithm in a way that made the correspondence proof relatively easy. But it was then encumbent upon us to prove that the abstract algorithm was actually a sorting algorithm, i.e., that it returns an ordered permutation of its input. In addition, in relating the abstract algorithm to the microcode it was necessary to prove several theorems about how the signed infinities are handled by the abstract algorithm. These theorems are especially interesting to prove.

Therefore, this paper presents the abstract sorting algorithm and the key theorems about it. We focus on the hardest of these theorems to prove, namely the treatment of positive infinities. Despite the general nature of these theorems – e.g., the absence of bounds on the lengths of the vectors being sorted or the size of the data – the reader is reminded that these theorems play a direct role in the very practical problem of the `5PEAK` microcode verification and are illustrative of the kind of general mathematics one must handle in code verification.

The abstract sorting algorithm sorts lists of "records" with integer keys. The algorithm is inspired by the operation of a comparator array, except that it uses an unlimited number of comparators.. The records are represented as cons pairs as constructed by cons (*other*, *data*), where where the *data* field represents the integer sort key, and the *other* field is arbitrary (but, in practice, contains the address from which the data was obtained). The basic systolic cycle of the general algorithm is captured by the function cstep.

DEFINITION:
cstep (*acc*)
=
if endp (*acc*) **then nil**
elseif endp (cdr (*acc*)) **then** *acc*
else cons (max-pair (cadr (*acc*), car (*acc*)),
cons (min-pair (cadr (*acc*), car (*acc*)),
cstep (cddr (*acc*))))

fi

where

DEFINITION:
max-pair (*pair1*, *pair2*)
=
if data (*pair1*) $\leq$ data (*pair2*) **then** *pair2*
else *pair1*
fi

DEFINITION:
min-pair (*pair1*, *pair2*)
=
if data (*pair1*) $\leq$ data (*pair2*) **then** *pair1*
else *pair2*
fi

The function cstep orders adjacent records in the accumulator *acc* pairwise, just as the comparator array orders C_n, P_n into P_n, C_{n+1} on each cycle.

Feeding the input vector into the unlimited resource comparator array is modeled by the function cfeed

DEFINITION:
cfeed (*lst*, *acc*)
=
if endp (*lst*) **then** *acc*
else cfeed (cdr (*lst*),
cstep (cons (car (*lst*), *acc*)))
fi

The function cfeed maintains an important invariant on the accumulator mentioned earlier in reference to the comparator array. If we number the elements of the accumulator, *acc*,

$$\mathrm{acc}_0, acc_1, \ldots, \mathrm{acc}_n$$

where acc_0 is the first element of the accumulator, then

$$\mathrm{acc}_i \geq \mathrm{acc}_j, \text{for } i \text{ even and } i < j.$$

That is, the even numbered elements dominate the elements to their right. Call this property Φ (*acc*). It is not difficult to prove that Φ is invariant under cfeed. That is, if an accumulator has property Φ and a list of records is fed into it

with cfeed then the result satisfies Φ. Since Φ (**nil**) holds, we can create an accumulator satisfying Φ by feeding an arbitrary list of records into the empty accumulator.

Furthermore, we can also prove that if a non-empty accumulator *acc* has property Φ, then the first element of *acc* is a maximal element and the result of applying cstep_ to cdr (*acc*) satisfies Φ. Thus, we can sort such an accumulator by 'draining' off the maxima while stepping the remainder.[1]

DEFINITION:
cdrain (n, *acc*)
=
if zp (n) **then** *acc*
else cons (car (*acc*),
cdrain (n − 1, cstep (cdr (*acc*))))
fi

The final sorting algorithm feeds the input data vector into an empty accumulator and then drains off the maxima.

DEFINITION:
csort (*lst*)
=
cdrain (len (*lst*), cfeed (*lst*, **nil**))

4 The Key Theorems

Given the foregoing claims about Φ it is not difficult to prove

> THEOREM: Ordered Permutation Property
> The function csort returns an ordered (weakly descending) permutation of its input.

To relate these abstractions to the microcode, we had to develop two other interesting and crucial properties. First, observe that in the definition of csort above the cfeed operation is done with the initial accumulator **nil**. But in the code, the corresponding operation is done with the peak and candidate value registers initialized to the most negative CAP integer. To prove that the code implements csort (in the sense described) we had to prove

[1] We define cdrain with a counter n because this is convenient for mapping operations from the actual fixed-size comparator array on the CAP to the unlimited resource comparator sorter.

THEOREM: Negative Infinity Property
Let *lst* be a list of records and *min* be one record, and suppose every element of *lst* dominates (i.e., has data greater than or equal to the data of)*min*. Let *minlst* be a list of n repetitions of *min*. Then cfeed (*lst*, *minlst*) is just append (cfeed (*lst*, **nil**), *minlst*).

This theorem tells us that if we initialize the comparator array to "negative infinities" as done on the CAP (i.e., to *minlst* where *min* is a record containing the most negative CAP integer) and then feed the input vector into it, the abstract result is the same as feeding the vector into an empty comparator array, as in our definition of csort, and then concatenating the "negative infinities" to the right. Since we are only interested in the first five elements, we can see that the negative infinities are irrelevant to the final answer if the input vector contains more than five elements.

The second interesting property concerns the fact that our csort uses the function cdrain while the second phase of the microcode performs this step by feeding in "positive infinities." We prove the following theorem to overcome this difference:

THEOREM: Positive Infinity Property
Let *acc* be a list of records satisfying Φ. Let *max* be a record that dominates every element of *acc*. Finally, let *maxlst* be a list of n repetitions of *max*, where n is an integer, $0 \leq n \leq |acc|$. Then cfeed (*maxlst*, *acc*) is append (*maxlst*, cdrain (n, *acc*)).

Note that the accumulator produced from **nil** by cfeed satisfies Φ and thus has the property required of *acc* in the theorem above. Furthermore, a list of n repetions of the "positive infinity" record has the property required of *maxlst* above. The theorem thus tells us that when the second phase of the CAP code feeds positive infinitives into the array the result is the same as concatenating positive infinities to the result of draining the comparators as specified in our definition of csort. Thus, at the conclusion of the second phase, the rightmost registers in the CAP array contain the answer computed by cdrain.

We find this relationship between cfeed and cdrain to be both surprising and beautiful.

5 Proof of the Positive Infinity Property

The two infinity properties are challenging to prove. We will briefly discuss our proof of the Positive Infinity Property. The problem is a familiar one to anyone who has proved theorems by induction: the theorem must be generalized. This problem is a mathematical one and is independent of the particular mechanized logic or mechanical theorem prover employed.

The theorem we wish to prove involves feeding a series of n max's into acc. What happens when you do that? The max's pile up (in reverse order) at the front and acc is stepped with cstep, except that odd/even parities of the elements of acc alternate because of the max's being added to the front. We leave to the reader the problem of discovering what goes wrong with an attempt to prove the theorem directly by induction, but dealing with these changing parities is one of the problems.

To prove the Positive Infinity Property we prove a stronger property by induction. We state the stronger property, Positive Infinity Property Generalized, below. But we motivate (and sketch the proof of) the property in the discussion below, where we explain how to strengthen the original property. The original property involves feeding a list of max's into an accumulator acc. We will generalize the theorem by generalizing both the list of max's and the accumulator. We start with the latter.

From the discussion above it is clear that the general state of the accumulator is not one merely satisfying Φ but one containing a pile of max's at the front and satisfying Φ. Thus, the accumulator should have the form append(s, acc'). At first it may appear sufficient to require that s be a list of max's and that append(s, acc') satisfy Φ, but we need to generalize further. In particular, we require that s be an ordered (weakly descending) list of records such that $|s|$ is even and every element of s dominates every element of acc'.

Note that under these conditions, if acc' satisfies Φ then so does the concatenation of s and acc'. The facts that $|s|$ is even and every element of s dominates every element of acc' allows us to distribute Φ over the concatenation, e.g., append(s, acc') has property Φ iff both s and acc' have the property. Note also that if $|s|$ is even, then cstep distributes over append also: the result of stepping the concatenation of s and acc' is the concatenation of the results of stepping s and stepping acc'. Such observations are crucial and we use them implicitly below.

So the general shape of the accumulator is append (s, acc') where s and acc' are as above.

Instead of feeding in a list of max's, we feed in an arbitrary list of records, lst, such that lst is ordered but weakly *ascending*, every element of lst dominates the elements of s and of acc', and $|lst| < |acc'|$.

To see why this version of the theorem is necessary, consider inductively proving a theorem involving the expression

$$\text{cfeed}(lst, \text{append}(s, acc'))$$

where lst, s and acc' have the properties required above.

In the induction step, lst is non-empty, i.e., is cons (mx, lst'). Consider what happens when we feed in the first element, mx, to the comparator. The function cfeed conses mx onto append(s, acc'), steps it, and recursively handles

lst'. That is, the expression above becomes

$$\text{cfeed}\,(lst', \text{cstep}\,(\text{cons}\,(mx, \text{append}\,(s, acc'))))$$

and we seek an induction hypothesis that will enable us to manipulate this expression further. But the inductive hypothesis will be of the form

$$\text{cfeed}\,(lst', \text{append}\,(\sigma, \alpha)),$$

for lst' and some σ and α satisfying our general conditions on lst, s and acc' above. Clearly, we must manipulate the cstep expression above, which we shall call ψ, into the append form.

Because $|lst| < |acc'|$ we can write acc' as $\text{cons}\,(a, acc'')$. Thus,

$$\begin{aligned}\psi &= \text{cstep}\,(\text{cons}\,(mx, \text{append}\,(s, acc'))) \\ &= \text{cstep}\,(\text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, acc'')))) \\ &= \text{cstep}\,(\text{append}\,(\text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, \mathbf{nil}))), acc''))\end{aligned}$$

Because $|s|$ is even, so is $|\text{cons}(mx, \text{append}(s, \text{cons}(a, \mathbf{nil})))|$. Thus, we can distribute cstep over the append to obtain

$$\begin{aligned}\psi = \text{append}\,&(\text{cstep}\,(\text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, \mathbf{nil})))), \\ &\text{cstep}\,(acc''))\end{aligned}$$

Since s is ordered, weakly descending, mx dominates everything in s and a is dominated by everything in s, the list $\text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, \mathbf{nil})))$ is ordered, weakly descending. Thus the first cstep expression above is a no-op.

$$\begin{aligned}\psi = \text{append}\,&(\text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, \mathbf{nil}))), \\ &\text{cstep}\,(acc''))\end{aligned}$$

Hence, ψ is in the form $\text{append}(\sigma, \alpha)$, where

$$\begin{aligned}\sigma &: \text{cons}\,(mx, \text{append}\,(s, \text{cons}\,(a, \mathbf{nil}))) \\ \alpha &: \text{cstep}\,(acc'')\end{aligned}$$

A little thought will show that these values of σ and α satisfy the conditions on s and acc' required by the theorem.

In short, an inductive proof of the following general theorem is straightforward, given the fairly subtle relationships between the conditions illustrated above.

> THEOREM: Positive Infinity Property Generalized
> Let acc' be a list of records satisfying Φ. Let lst be a list of records such that $|lst| < |acc'|$ and suppose that lst is ordered weakly ascending. Let s be a list of records such that $|s|$ is even and s is ordered weakly descending. Finally, suppose every element of lst dominates every element of s and of acc' and that every element of s dominates every element of acc'. Then

$$\begin{array}{l}\text{cfeed}\,(lst, \text{append}\,(s, acc')) \\ \quad = \\ \text{append}\,(\text{reverse}\,(lst), s, \text{cdrain}\,(|lst|, acc')).\end{array}$$

Note that our Positive Infinity Property follows from the one above, if we let s be **nil**, acc' be acc and lst be a list of n max's.

6 A Tour of the ACL2 Proof Script

Proofs of all three of the key theorems noted here have been checked with the ACL2 theorem prover. Ninety-five ACL2 definitions and theorems are involved in our proof of these theorems. This includes the definitions necessary to define all of the concepts. The ACL2 input script or "book" is http://www.cs.utexas.edu/users/moore/publications/csort/csort.lisp. We give a brief sketch of the book here. The book is divided into six "chapters."

Chapter 1 deals with elementary list processing and is completely independent of the specifics of the comparator sort problem. It defines functions for retrieving the first n elements of a list and for producing a list of n repetitions of an element, and it defines the predicate that determines whether one list is a permutation of another. The chapter then proves fundamental properties of several primitive ACL2 functions and these functions, including

- the concatenation function is associative,
- the length of the concatenation of two lists is the sum of their lengths,
- the first n element of a list of length n is the list itself, and
- the reverse of n repetitions of an element is just n repetitions of the element.

The most important contribution of this chapter is that it establishes that the permutation predicate is an equivalence relation and that it is a congruence relation for certain Lisp primitives such as list membership, concatenation, and length. Here we will denote that a is a permutation of b by "$a \simeq b$". When we say that the permutation predicate is a congruence relation for (the second argument of) list membership, we mean "$a \simeq b \rightarrow (x \in a) \leftrightarrow (x \in b)$".

ACL2 supports congruence-based rewriting. When ACL2 rewrites an expression it does so in a context in which it is trying to maintain some given equivalence relation. Generally, at the top-level of a formula, it rewrites to maintain propositional equivalence. Because of the above congruence relation, when ACL2 rewrites an expression like "$\gamma \in \alpha$" to maintain propositional equivalence ("$\leftrightarrow$") it can rewrite α to maintain the permutation relation ("$\simeq$").

How does ACL2 rewrite maintaining "$\simeq$"? The answer is that it uses rewrite rules that use "$\simeq$" as their top-level predicate. For example, the theorem that

reverse(x) $\simeq x$ can be so used as a rewrite rule in the second argument of "$\in$". Thus, "$e \in$ reverse(x)" rewrites to "$e \in x$". This rewrite rule about reverse (modulo $\simeq$) is included in the first chapter of our book. In all, Chapter 1 contains 37 definitions and theorems.

Chapter 2 deals with the idea of ordering lists of pairs by the "data" component. It defines the function "data" and the predicate "ordered" and also defines two other predicates. The first ("all-gte") checks that one pair dominates all the pairs in a given list, in the sense that the pair's data field is greater than or equal to that of each of the other pairs. The second ("all-all-gte") checks that every pair in one list dominates all the pairs in another. These predicates are used in our formalizations of the two infinity properties.

The chapter then lists about 20 theorems about these functions and predicates, including,

- that permutation is a congruence relation for all-gte and all-all-gte, e.g., that $a \simeq b \rightarrow \text{all-gte}\,(p, a) \leftrightarrow \text{all-gte}\,(p, b)$.
- that the concatenation of two lists is ordered precisely when the two lists are ordered and all the elements of the first dominate those of the second,
- that a pair dominates the elements of the concatenation of two lists precisely when it dominates all the elements of each list, and
- that the list consisting of n repetitions of an element is ordered.

A total of 25 events are in this chapter.

Chapter 3 contains the six events defining min-pair, max-pair, cstep, cfeed, cdrain and csort.

Chapter 4 establishes the basic properties of the above-mentioned functions, including that cstep, cfeed, and cdrain produce permutations of their arguments, the corollaries that the lengths of their outputs are suitably related to the lengths of their inputs, and that cstep has these three properties:

- cstep distributes over the concatenation of two lists if the first list has even length,
- cstep distributes over the concatenation of two lists if the second list is ordered and is dominated by the elements of the first list,
- cstep is a no-op on ordered lists.

Thirteen theorems are in this chapter.

In Chapter 5 we are concerned with the invariant ϕ. We define it and prove

- $\phi(\text{cdr}(acc)) \rightarrow \phi(\text{cstep}(acc))$, and
- $\phi(acc) \rightarrow \phi(\text{cfeed}\,(lst, acc))$.

Two other lemmas are proved to help ACL2 to find the proofs of these two theorems.

Finally, in Chapter 6 we prove the three theorems discussed in this paper. The theorem that csort produces an ordered permutation of its input is decomposed into two parts. The permutation part, csort(acc) $\simeq$ acc, is trivial, given the work done in Chapter 4. The ordered part is ordered(csort(lst)) and is proved using the lemma:

- If n is a natural number such that $n \leq |acc|$ and acc has property ϕ, then ordered(firstn$(2 + n,$ cdrain$(n, acc))$).

The Positive Infinity Property is proved using the lemma below.

- Suppose data $(p_1) \geq$ data (p_2). Suppose s is an ordered list of even length, p_1 dominates every element of s, and every element of s dominates p_2. Then

$$\text{cstep (cons } (p_1, \text{append } (s, \text{cons } (p_2, acc)))) \\ = \text{cons } (p_1, \text{append } (s, \text{cons } (p_2, \text{cstep } (acc)))).$$

This lemma is the key simplification step in the proof discussed above of Positive Infinity Property Generalized, which is the next theorem proved in this chapter. It is necessary to tell ACL2 to use the particular induction scheme used in our discussion. The Positive Infinity Property is then proved by instantiation.

The Negative Infinity Property relies on a similar, inductively proved generalization:

- Suppose acc is ordered. Suppose further that every element of lst dominates every element of acc and that every element of s dominates acc. Then cfeed $(lst,$ append $(s, acc))$ = append (cfeed $(lst, s), acc)$.

It takes about 25 seconds to prove all of theorems in all of the chapters. This measurement is taken on a 200 MHz Sun Microsystems Ultra-2.

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press: New York, 1979.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*, Academic Press: London, 1997.

[3] B. Brock, M. Kaufmann and J S. Moore, "ACL2 Theorems about Commercial Microprocessors," in M. Srivas and A. Camilleri (eds.) *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275–293, 1996.

[4] M. J. Curey, P. M. Hansen, and C. D. Thompson, "Sorting Records in VLSI," in L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel (eds) *Algorithmically Specialized Parallel Computers*, Academic Press, New York, NY, pp. 27–36, 1985.

[5] S. Gilfeather, J. Gehman, and C. Harrison. Architecture of a Complex Arithmetic Processor for Communication Signal Processsing in *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, **2296** *Advanced Signal Processing: Algorithms, Architectures, and Implementations V*, July, 1994, pp. 624–625.

[6] M. Kaufmann and J Strother Moore "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp,"*IEEE Transactions on Software Engineering*, **23**(4), pp. 203–213, April, 1997.

[7] M. Kaufmann and J Strother Moore "A Precise Description of the ACL2 Logic," http://www.cs.utexas.edu/users/moore/publications/-km97a.ps.Z, April, 1998.

[8] G. Miranker, L. Tang, and C. K. Wong, A "Zero-Time' VLSI Sorter, *IBM J. Res. Develop*, **27**(2), March, 1983, pp. 140–148.

[9] G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.